

Décimo octava Sesión

Metodologías y Técnicas de Programación II

**Programación Orientada a
Objeto (POO)
C++**

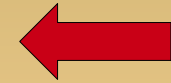
Herencia II

Estado del Programa

Introducción a la POO

Historia de la Programación
Conceptos de POO

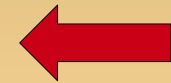
C++
Mi primera Clase



Repaso de Conceptos

Estándares de Programación
Punteros y Memoria

E/S
Control y Operadores



Clases y Objetos en C++

Uso y aplicación
Constructores
Constantes e "inline"

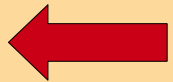
Funciones Amigas
Sobrecarga de Funciones



Sobrecarga

De Operadores

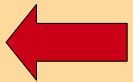
Creación Dinámica de Objetos



Herencia.

Tipos de Visibilidad

Herencia Múltiple



Polimorfismo

Funciones Virtuales

Polimorfismo y Sobrecarga.

Plantillas

Contenedores

Iteradores

18.1 Repaso

Herencia y Composición

Formas de reutilización sistemática. Mejor. Más concienzudas.

Composición: Creamos objetos de la clase existente dentro de la nueva clase que estamos creando.

La nueva clase **está compuesta** por objetos de clases existentes.

Herencia: Se toma la forma de la clase existente y se añade código sin modificar la clase existente.

La nueva clase **se crea como un tipo** de la existente.
Casi todo el trabajo lo realiza el compilador.

La herencia es una propiedad de la Programación Orientada a Objeto.

Nos permite:

Crear nuevas clases a partir de clases existentes.

Conservando las propiedades de la clase original.

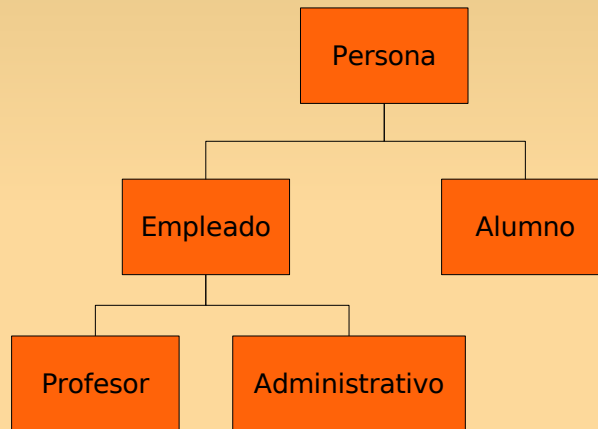
Añadir nuevos métodos y atributos a la clase original.

18.1 Repaso

Herencia

Cuando una clase deriva de más de una clase base: **Herencia Múltiple.**

Este tipo de relaciones nos permite crear **Jerarquías de Clases.**



`class <clase_derivada> :`

`[public|private] <base1> [, [public|private] <base2>] {};`

Clase base: La clase base es la clase ya creada, de la que se hereda.

También se la denomina clase madre o superclase.

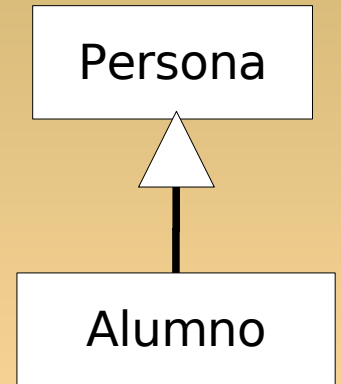
Clase derivada: es la clase que se crea a partir de la clase base. Se dice que es la clase que hereda.

También se la denomina clase hija o subclase.

18.2 Repaso

Control de Acceso en la Herencia

```
class <clase_derivada> :  
[public|private] <base1> [, [public|private] <base2>] {};
```



Tipo de Dato Clase Base	Herencia con public	Herencia con private	Otros
Private	No accesible directamente	No accesible directamente	No accesible
Protected	Protected	Private	No accesible
Public	Public	Private	Accesible (. ó ->)

18.2 Herencia

Herencia: Miembros que no se heredan automáticamente

La clase derivada hereda todos los atributos y todos los métodos, excepto:

- **Constructores**
- **Destructor**
- **Operador de asignación**

Constructores y destructor:

Si en la subclase no están definidos, se crea un constructor por defecto y un destructor, aunque sin que hagan nada en concreto.

Operador de asignación:

Si en la subclase no está definido, se crea uno por defecto que se basa en el operador de asignación de la superclase.

Se deben crear los constructores y el destructor en la subclase.

Se debe definir el operador de asignación en la subclase.

18.2 Herencia

Herencia: Constructores y Lista de Inicialización

En C++ es muy importante la correcta inicialización de los objetos y variables.

Cuando se crea un objeto el compilador garantiza la ejecución de todos los constructores para cada uno de los subobjetos.

¿Qué ocurre..?

- Si uno de los subobjetos no tiene un constructor por defecto.
- Si queremos cambiar los parámetros por defecto de un constructor.

La solución:

- Ejecutamos nosotros la llamada al constructor.
- Utilizamos los inicializadores o listas de inicialización.

Esto ya lo hemos usado (solo que con tipos predefinidos):

```
Alumno::Alumno(int nota, int edad) : nota_(nota), edad_(edad) {};  
Alumno::Alumno(int nota, int edad) : Persona(edad), nota_(nota) {};
```

18.2 Herencia

Constructores

```
class ClaseBase
{
    protected:
        int b1 ;
        int b2 ;
    public:
        int b3 ;
        ClaseBase(int a=0,int b=0,int c=0);
        ...
};
class ClaseDerivada : public ClaseBase
{
    private:
        float s1 ;
        char s2 ;
    public:
        ClaseDerivada(float d, char e);
};
```

Cuando se llama al constructor de la clase derivada, el compilador ejecuta el constructor por defecto de la clase Base, a no ser que especifiquemos uno en concreto.

```
ClaseDerivada d(8.2, 'A');
```

```
b1 - 0    s1 - 8.2
b2 - 0    s2 - A
b3 - 0
```

18.2 Herencia

Constructores

```
class ClaseBase
{
    protected:
        int b1 ;
        int b2 ;
    public:
        int b3 ;
        ClaseBase(int a=0,int b=0,int c=0);
        ...
};

class ClaseDerivada : public ClaseBase
{
    private:
        float s1 ;
        char s2 ;
    public:
        ClaseDerivada(int a, int b, int c,
                       float d, char e);
};
```

Si queremos que los atributos heredados tomen un valor determinado:

```
ClaseDerivada :: ClaseDerivada
(int a,int b,int c,float d,char e)
: ClaseBase(a,b,c)
{
    s1 = d;
    s2 = e;
};
```

```
ClaseDerivada d(6,7,48.2, 'A');
```

```
b1 - 6    s1 - 8.2
b2 - 7    s2 - A
b3 - 4
```

18.2 Herencia

Herencia: Llamadas automáticas al destructor

A menudo es necesario realizar llamadas explícitas a los constructores en la inicialización como hemos visto.

Nunca será necesario realizar una llamada explícita a los destructores:

- Sólo existe un destructor para cada clase.
- No tiene parámetros.

El compilador asegura que **todos los destructores son llamados**.

Esto significa que todos los destructores de la jerarquía, desde el destructor de la clase derivada y retrocediendo hasta la raíz, serán ejecutados.

18.2 Herencia

Orden de llamada de constructores y destructores

```
#define CLASS(ID) class ID { \  
public: \  
ID(int) {out<< #ID " constructor\n";}\  
~ID() {out << #ID " destructor\n"; } \  
};  
CLASS(Base1);  
CLASS(Member1);  
CLASS(Member2);  
CLASS(Member3);  
class Derived1 : public Base1  
{  
    Member1 m1;  
    Member2 m2;  
public:  
    Derived1(int) : m2(1), m1(2),  
    Base1(3) {
```

```
        out << "Derived1 constructor\n";  
    }  
    ~Derived1() {  
        out << "Derived1 destructor\n";  
    }  
};  
class Derived2 : public Derived1  
{  
    Member3 m3;  
public:  
    Derived2() : m3(1), Derived1(2)  
    {  
        out << "Derived2 constructor\n";  
    }  
    ~Derived2()  
    {  
        out << "Derived2 destructor\n";  
    }  
};
```

18.2 Herencia

Orden de llamada de constructores y destructores

```
int main()  
{  
    Derived2 d2;  
}
```

SALIDA POR PANTALLA - ¿Es Correcta?

=====

Base1 constructor

Member1 constructor

Member2 constructor

Derived1 constructor

Member3 constructor

Derived2 constructor

Derived2 destructor

Member3 destructor

Derived1 destructor

Member2 destructor

Member1 destructor

Base1 destructor

18.2 Herencia

Orden de llamada de constructores y destructores

El orden de las llamadas al constructor para los objetos miembro no se ve afectado por el orden de las llamadas en la lista de inicializadores de un constructor.

Es determinado por el orden en que los objetos miembros son declarados en la clase.

Si pudiéramos cambiar el orden del constructor en la lista de inicializadores de un constructor, podríamos tener dos secuencias diferentes de llamada en dos constructores diferentes, pero el destructor no sabría como invertir el orden para llamarse correctamente y nos encontraríamos con problemas de dependencias.

18.2 Herencia

Orden de llamada de constructores

Cuando se crea un objeto de la clase derivada, ocurre lo siguiente:

1º Se invoca el constructor de la superclase. La invocación del constructor de la superclase se realiza con los argumentos que se especifiquen. Si no hay argumentos, se usa el constructor predeterminado.

2º Se ejecuta el constructor de la clase derivada.



18.2 Herencia

Orden de llamada de destructores

Cuando se destruye un objeto de la clase derivada, ocurre lo siguiente:

1º Se invoca el destructor de la clas derivada.

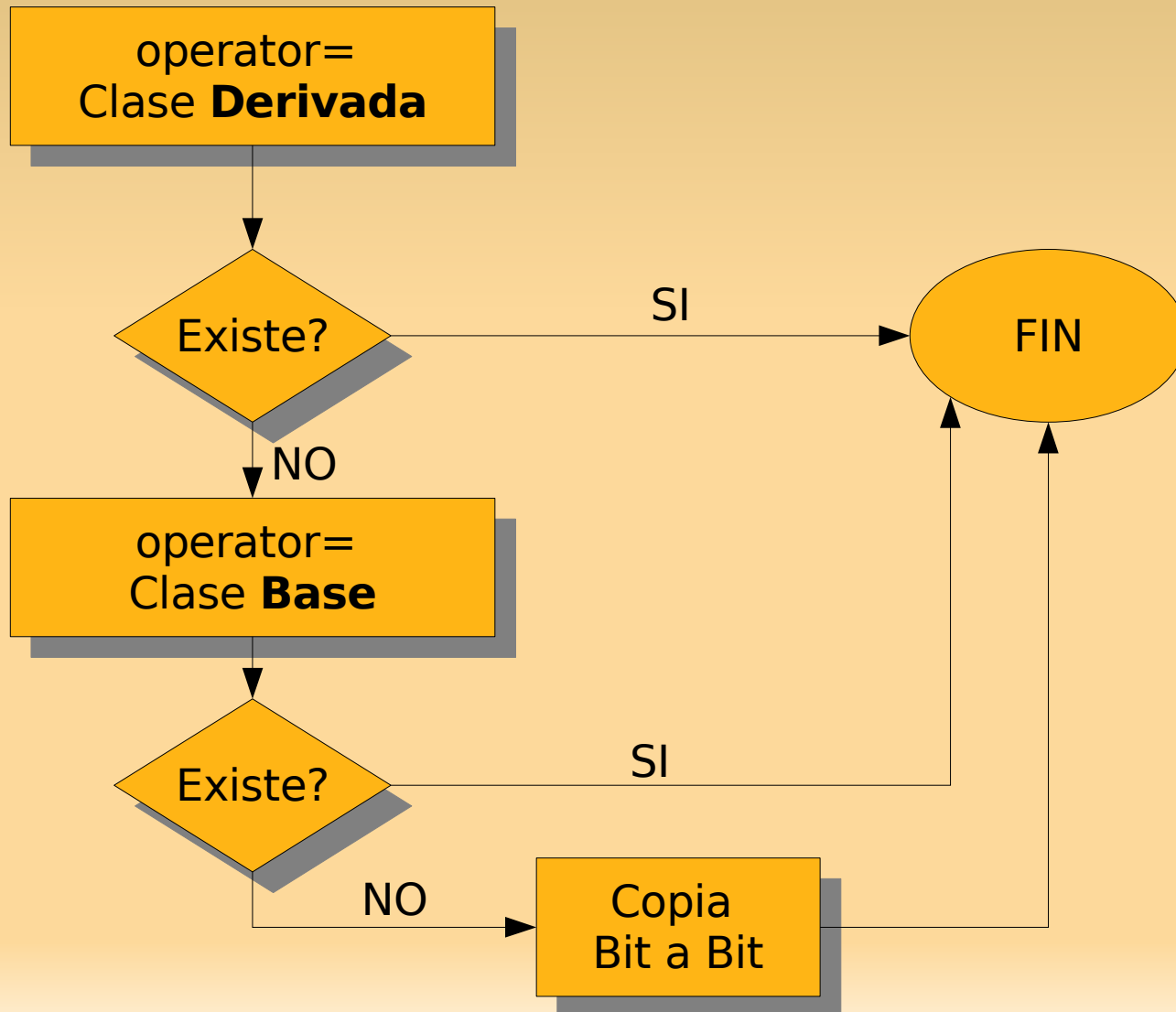
2º Se ejecuta el destructor de la clase base.



17.2 Herencia

Operador Asignación

Recordemos que el operador asignación NO se hereda.



18.2 Herencia

Operador Asignación

```
ClaseBase& ClaseBase ::  
operator = ( ClaseBase c )  
{  
    b1 = c.b1;  
    b2 = c.b2;  
    b3 = c.b3;  
    return *this;  
};  
ClaseDerivada& ClaseDerivada ::  
    operator = ( ClaseDerivada d )  
{  
    s1 = d.s1;  
    s2 = d.s2;  
    return *this;  
};  
ClaseDerivada ob1 ( 6,7,4,8.2, 'f' );  
ClaseDerivada ob2 ( 0,0,0, 0, 'h' );  
ob2 = ob1;
```

¿Qué método se ejecuta?

```
b1 - 6    s1 - 8.2  
b2 - 7    s2 - f  
b3 - 4
```

```
b1 - 0    s1 - 0  
b2 - 0    s2 - h  
b3 - 0
```

```
b1 - 0    s1 - 8.2  
b2 - 0    s2 - f  
b3 - 0
```

18.2 Herencia

Operador Asignación

```
ClaseBase& ClaseBase ::  
operator = ( ClaseBase c )  
{  
    b1 = c.b1;  
    b2 = c.b2;  
    b3 = c.b3;  
    return *this;  
};  
  
ClaseDerivada& ClaseDerivada ::  
operator = ( ClaseDerivada d )  
{  
    b1 = d.b1;  
    b2 = d.b2;  
    b3 = d.b3;  
    s1 = d.s1;  
    s2 = d.s2;  
    return *this;  
};
```

Una Solución:

Podemos añadir al operador asignación de la clase derivada cada una de las asignaciones que “nos faltan” en el anterior caso.

¿Qué pasa si tenemos 120 atributos en la clase base?

¿Me tengo que preocupar de si se añaden o quitan atributos a una clase que puede que ni siquiera mantengamos nosotros?

18.2 Herencia

Operador Asignación

```
ClaseBase& ClaseBase ::  
operator = ( ClaseBase c )  
{  
    b1 = c.b1;  
    b2 = c.b2;  
    b3 = c.b3;  
    return *this;  
};  
ClaseDerivada& ClaseDerivada ::  
    operator = ( ClaseDerivada d )  
{  
    ClaseBase::operator=(d);  
    s1 = d.s1;  
    s2 = d.s2;  
    return *this;  
};  
ClaseDerivada ob1 ( 6,7,4,8.2, 'f' );  
ClaseDerivada ob2 ( 0,0,0, 0, 'h' );  
ob2 = ob1;
```

Ejecutamos el operador asignación de la superclase.

```
ob1  b1 - 6   s1 - 8.2  
      b2 - 7   s2 - f  
      b3 - 4
```

```
ob2  b1 - 0   s1 - 0  
      b2 - 0   s2 - h  
      b3 - 0
```

```
ob2  b1 - 6   s1 - 8.2  
      b2 - 7   s2 - f  
      b3 - 4
```

18.2 Herencia

Ejemplo de Combinación de Composición y Herencia

```
class A
{
    int i;
public:
    A(int ii) : i(ii) {};
    ~A() {};
    void f() const {};
};
```

```
class B
{
    int i;
public:
    B(int ii) : i(ii) {};
    ~B() {};
    void f() const {};
};
```

```
class C : public B // Herencia
{
    A a;          // Composición
public:
    C(int ii) : B(ii), a(ii) {};
    ~C() {}; // Llama a ~A() and ~B()
    void f() const // Redefinición
    {
        a.f();
        B::f();
    }
};

int main()
{
    C c(47);
    c.f();
}
```

18.2 Herencia

Ejemplo de Herencia

```
Class Alumno : public Persona
{
private:
    int curso;
public:
    Alumno(char * , int = 0, char *,
           char * , int );
    Alumno& operator=( Alumno &);
    ~Alumno ();    // Destructor
    int mcurso ();
    void mcurso (int );
};
//-----
int Alumno :: mcurso ()
{
    return curso;
}
```

```
Alumno & Alumno ::operator= (Alumno a)
{
    Persona :: operator = (a);
    curso = a.curso;
    return *this;
}
Alumno :: Alumno (char * n, int e, char *
nom, char * ape , int c )
: Persona (n, e, nom, ape)
{
    curso = c;
}

void Alumno :: mcurso (int c)
{
    curso = c ;
}
```

18.2 Herencia

Herencia y Redefinición de Métodos

Vimos que con la Herencia podemos:

- Heredar y reutilizar atributos y métodos.
- Ampliar atributos o métodos.
- Redefinir Métodos.

¿Tiene sentido redefinir atributos?

En la clase derivada se puede redefinir algún método ya definido en la clase base: redefinición o superposición de métodos.

Para redefinir un método en la subclase, basta con declarar una función miembro con el mismo nombre.

Por ejemplo:

- El método mostrar() de Persona no tiene sentido para Alumno.
- Redefinimos el método mostrar() en Alumno.

18.2 Herencia

Redefinición de Métodos

```
void Persona :: mostrar()
{
    cout << nif;
    cout << "Nombre: " << nombre;
}

void Alumno :: mostrar()
{
    Persona::mostrar();
    cout << "Curso: " << curso;
}

int main()
{
    Persona p1("89411N", 33,
              "Luis", "Fernan");
    Alumno alum ("77777R", 20,
                "Ana" , "Ruiz", 3 );

    p1.mostrar();
    alum.mostrar();
}
```

En el constructor de alum se llamará al constructor de Persona (porque así lo programamos).

alum y p1 quedan adecuadamente inicializados.

La salida es:

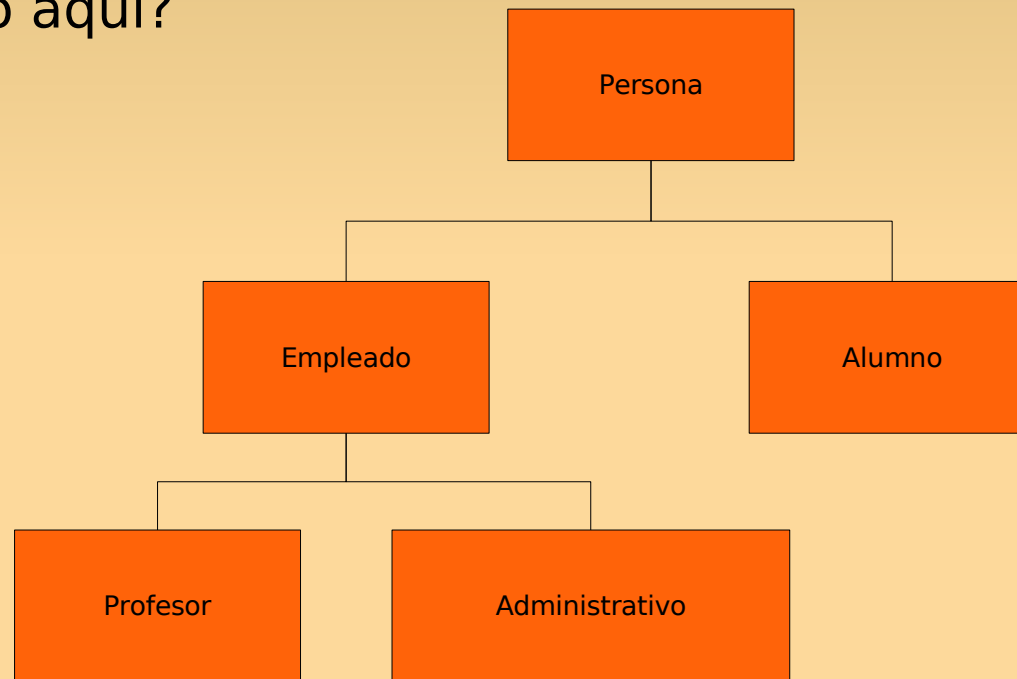
```
89411N Nombre: Luis
77777R Nombre: Ana Curso: 3
```

18.2 Herencia

Herencia, Jerarquías de clases y Redefinición de Métodos

Hemos visto un ejemplo sencillo: Clase Base (Persona) y Derivada (Alumno)

¿Cómo funciona todo esto aquí?



Hay diferentes situaciones dependiendo de cómo puede estar implementado en método mostrar:

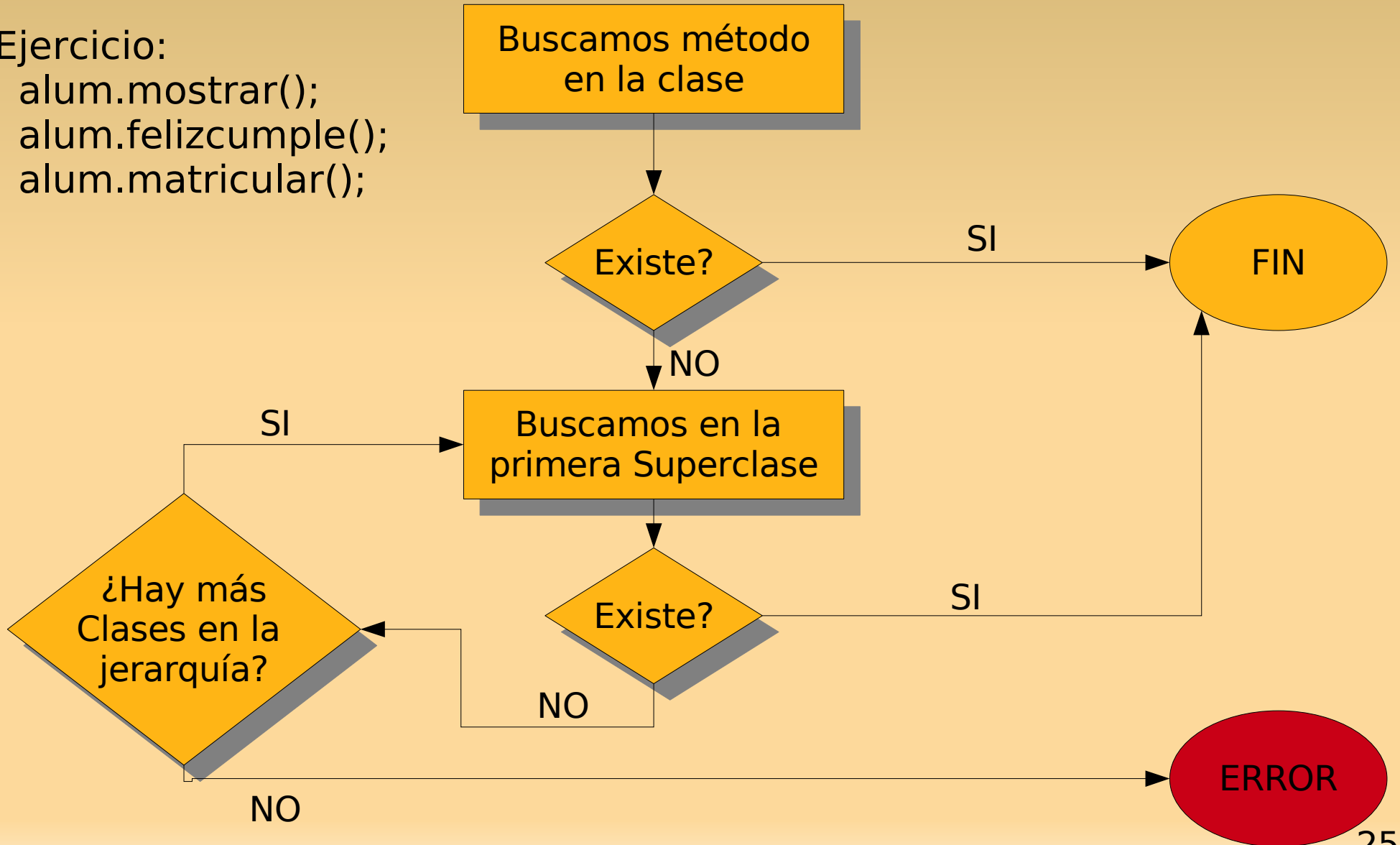
```
admin.mostrar(); // Objeto de la clase Administrativo...
```

17.2 Herencia

Pasos de Mensaje con Herencia

Ejercicio:

```
alum.mostrar();  
alum.felizcumple();  
alum.matricular();
```



18.2 Herencia

Tipos de Vinculación

Vinculación estática: se trata del intento de vincular el mensaje con el método correspondiente en tiempo de compilación.

Si se produce error de vinculación, será en tiempo de compilación

Vinculación dinámica: la vinculación entre mensaje y método se realiza en tiempo de ejecución.

Si se produce error de vinculación, será en tiempo de ejecución

18.2 Herencia

Problemas con la Vinculación

```
void Persona :: felizcumple()
{
    edad++;
    cout << "Felicidades!!";
    mostrar();
}

int main()
{
    Alumno alum ("77777R", 20,
                "Ana" , "Ruiz", 3 );
    alum.felizcumple();
}
```

El método felizcumple() no está definido en Alumno.

Se busca en persona, y se encuentra.

El mensaje mostrar() se vincula con la clase Persona en lugar de con la clase Alumno.

La salida es:

```
Felicidades!!
77777R Nombre: Ana
```

(*) Nos falta el curso.

18.3 Ejercicios

1.- **Compilar y ejecutar el siguiente código. Explicar lo que ocurre.**

```
class Base
{
public:
    int f() const
    {
        cout << "Base::f()\n";
        return 1;
    }
    int f(string) const { return 1; }
    void g() {}
};
class Derived1 : public Base
{
public:
    void g() const {}
};
class Derived2 : public Base
{
public:
    int f() const
    {
        cout << "Derived2::f()\n";
        return 2;
    }
};
```

```
class Derived3 : public Base
{
public:
    void f() const // Atentos
        { cout << "Derived3::f()\n"; }
};
class Derived4 : public Base
{
public:
    // Hay cambios??
    int f(int) const {
        cout << "Derived4::f()\n";
        return 4;
    }
};
string s("hello");
Derived1 d1;
int x = d1.f();
d1.f(s);
Derived2 d2;
x = d2.f();
Derived3 d3;
Derived4 d4;
x = d4.f(1);
```

18.3 Ejercicios

2.- Compila y ejecuta el código de las páginas 11 y 12. Comprueba si la salida por pantalla es correcta. Comenta lo que ocurre.

3.- Crea dos clases, A y B, con constructores por defecto que muestren por pantalla traza de que han sido llamados. Una nueva clase llamada C que hereda de A, y cree un objeto miembro B dentro de C, pero no cree un constructor para C. Crea un objeto de la clase C en main() y observa los resultados.

4.- Crea una jerarquía de clases de tres niveles con constructores por defecto y con destructores, ambos notificándose utilizando cout. Verificar que el objeto más alto de la jerarquía, los tres constructores y destructores son ejecutados automáticamente. Explicar el orden en que han sido realizados.

5.- Describe estos conceptos de forma breve:

- a) Herencia.
- b) Composición.
- c) Herencia Múltiple.
- d) Clase Base
- e) Clase Derivada

6.- Implementa las clases:

Punto, Círculo y Cilindro utilizando la Herencia.

Todos deben tener constructor basado en clase base, destructor y mostrar().