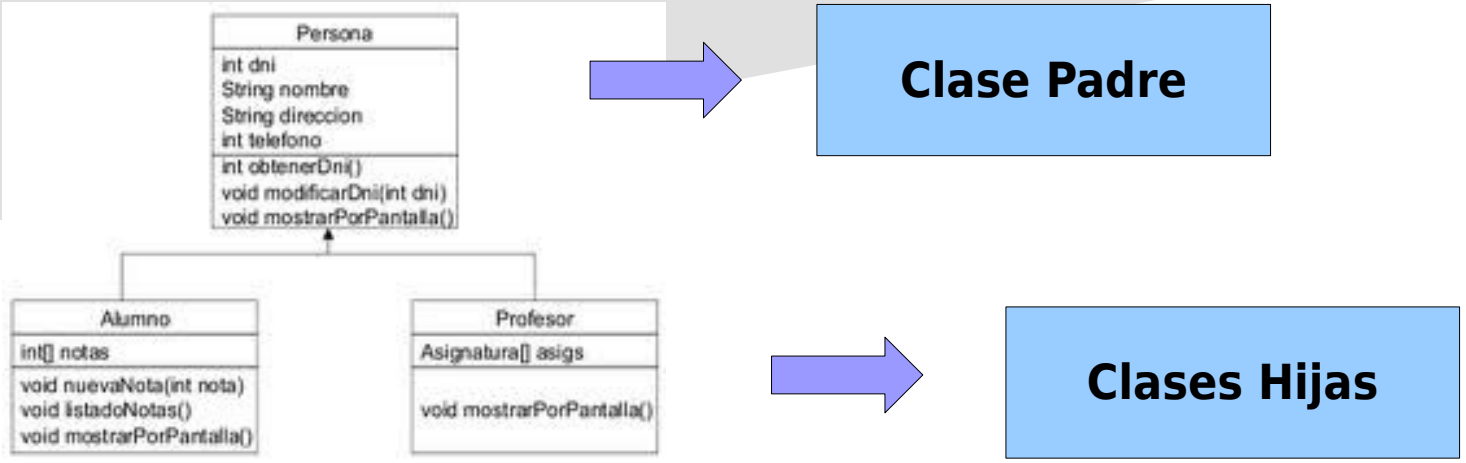
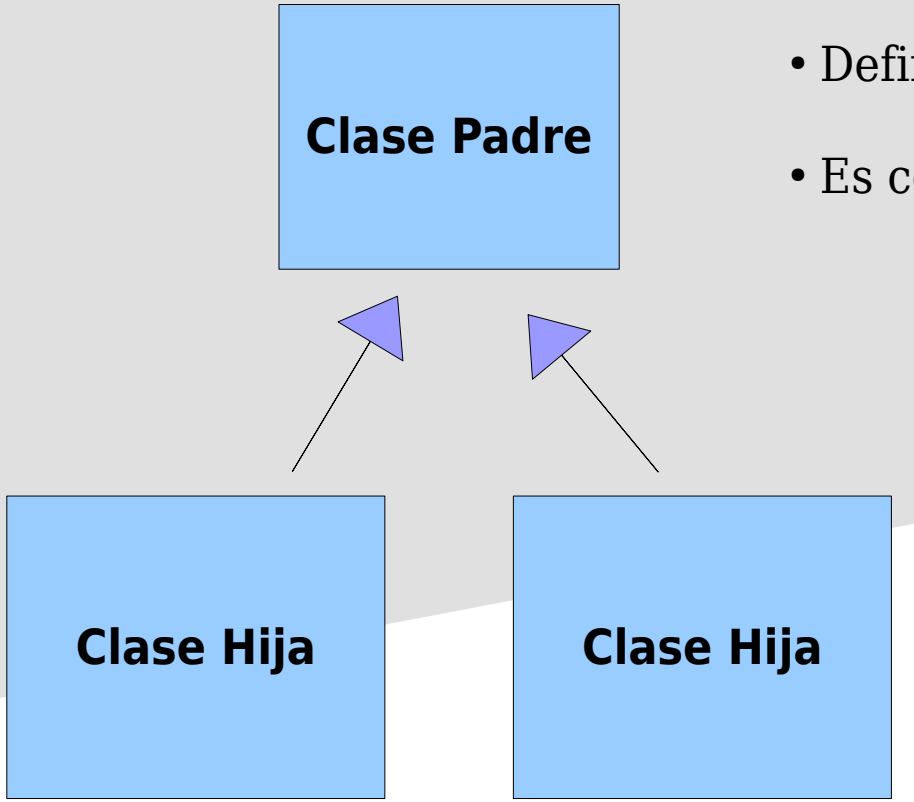
Herencia

- La reusabilidad puede lograrse mediante ***herencia***.
- Un comportamiento definido en una *superclase* es heredado por sus *subclases*.
- Las subclases extienden la funcionalidad heredada
- Esto nos permite definir la mayor cantidad de funcionalidades y atributos y luego reutilizarlas.



Herencia

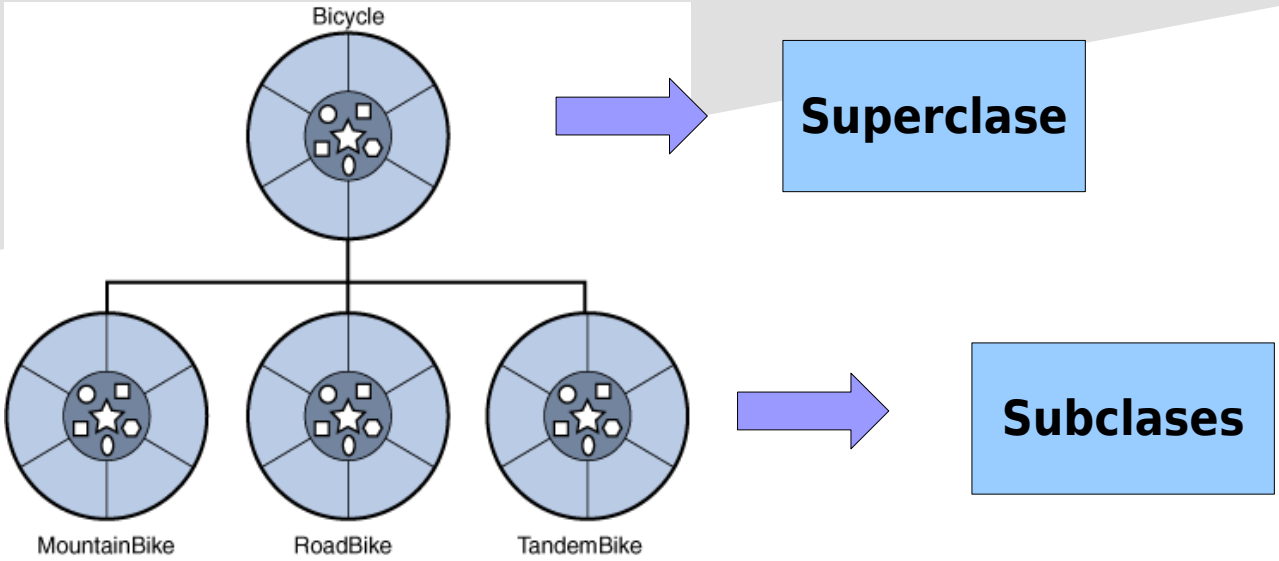


- Definimos atributos y comportamientos comunes.
- Es considerada como un tipo de generalización.

- Reutilizamos todo lo definido en la clase padre.
- Podemos ampliar atributos y comportamientos o *redifinirlos.*

Herencia

- Una clase hereda de su padre todos los atributos y métodos públicos y protegidos.
- Los constructores no son heredados pero pueden invocarse.
- Los métodos y atributos privados no se heredan.
- Si la subclase esta en el mismo paquete que la clase padre, también se heredan los miembros default.



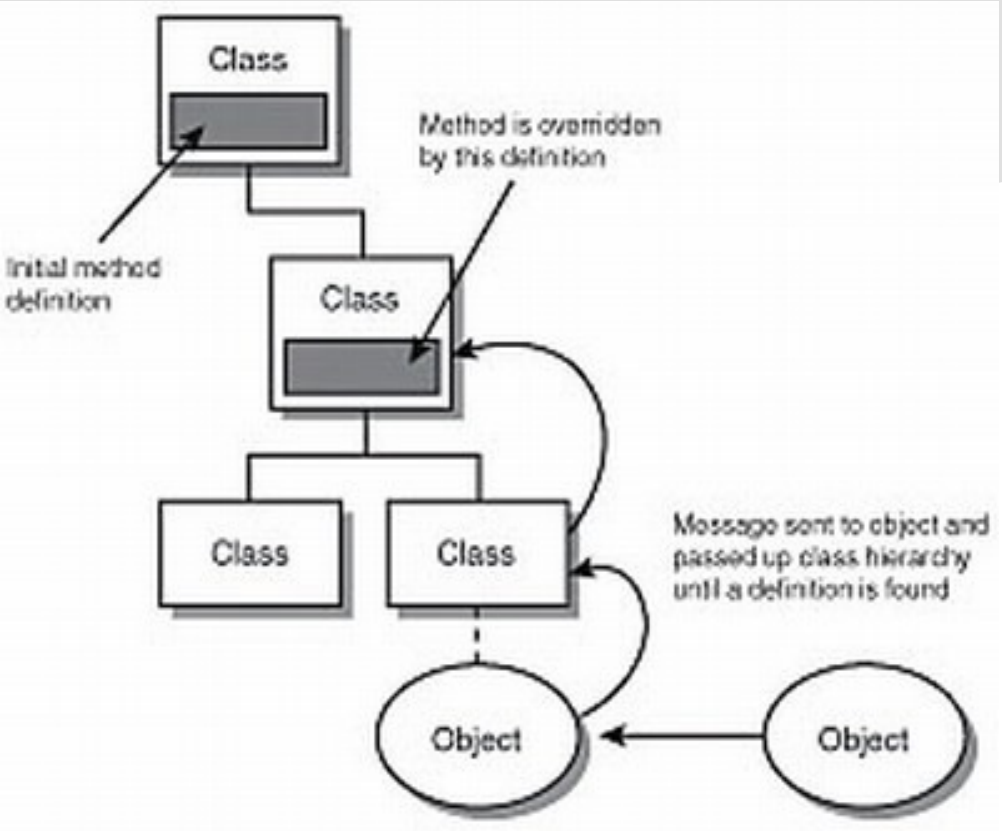
Herencia en C++

- Para representar la herencia utilizaremos : luego del nombre de la clase

```
1   #ifndef ALUMNO_H
2   #define ALUMNO_H
3   #include "persona.h"
4
5   class Alumno : public Persona
6   {
7   public:
8       Alumno();
9   };
10
11  #endif // ALUMNO_H
12
```

Sobreescritura de métodos

- Se da en el contexto de relaciones de herencia.
- Consiste en reescribir la implementación de un método de *instancia*, con su mismo nombre y argumentos (*firma del método*).
- Si una clase hija sobreescribe un método de su clase padre, entonces ocultará al mismo.



Sobreescritura de métodos

Un ejemplo

```persona.h
#ifndef PERSONA_H
#define PERSONA_H


class Persona
{
private:
    long dni;
    char * nombre;
public:
    Persona(long dni);
    long getDni();
    char * getNombre();
    void setNombre(char * nombre);
    void virtual mostrarInfo();
};

#endif // PERSONA_H
```

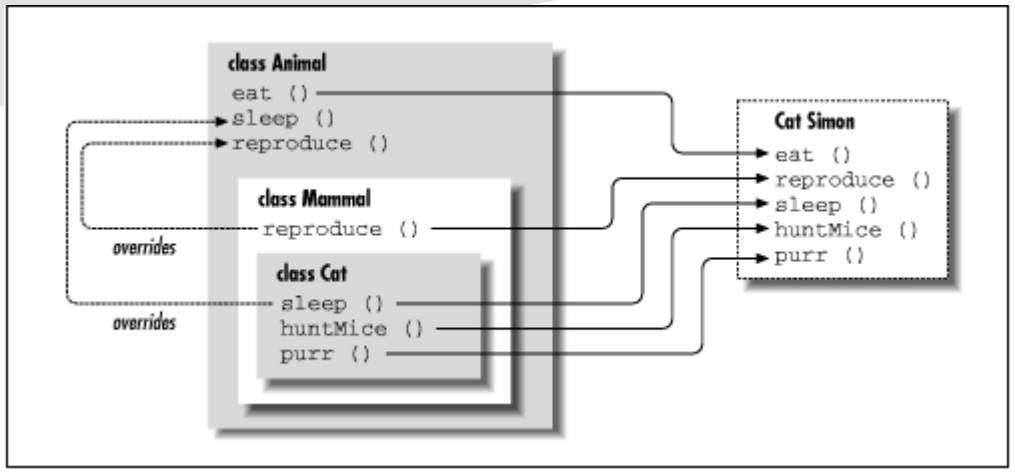
```alumno.h
#ifndef ALUMNO_H
#define ALUMNO_H
#include "persona.h"

class Alumno : public Persona
{
public:
    Alumno(long dni);
    void mostrarInfo();
};

#endif // ALUMNO_H
```

Sobreescritura de métodos

- La lista de argumentos debe ser idéntica en tipos y orden en ambos métodos.
- Los modificadores de acceso no pueden ser más restrictivos en la clase padre que en la subclase.
- Obtendremos un error de compilación si intentamos cambiar un método de instancia en una superclase a método de instancia en la subclase y viceversa.



Llamar a métodos de la clase padre

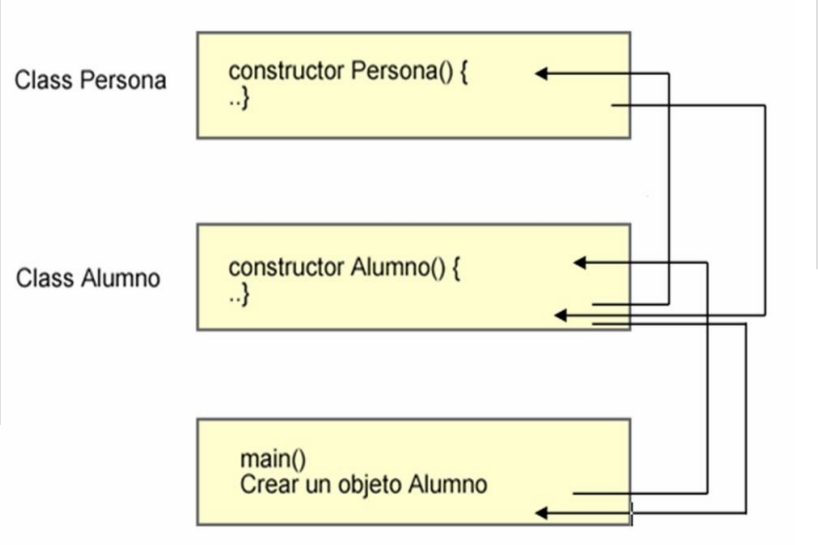
• Podemos llamar a un método de la clase padre escribiendo la clase padre :: el método :

```
void Alumno::mostrarInfo(){
    std::cout << "Alumno" << " ";
    Persona::mostrarInfo();
}
```

Encadenamiento de constructores

- Debido a que los constructores son un tipo especial de métodos, también pueden sobrecargarse.

- Esto nos permite asegurar mínimamente como se inicialicen los objetos, no importando que constructor se utilice.



Encadenamiento de constructores

Un ejemplo

```
3
4   Alumno::Alumno(long dni) : Persona(dni) {
5
6   }
7
```

Aseguramos que mínimanente como se creen los objetos tengan un estado mínimo

Downcasting y Upcasting de Objetos

Un objeto de un tipo se puede tratar como objeto de otro tipo siempre que la clase fuente y la clase destino estén relacionadas por herencia. En c++ un puntero puede ser tipado por su clase general pero apuntar a una subclase.

Upcasting: conversión de clase derivada a la clase base, se hace implícitamente.

```
Persona * p = new Alumno(555555);
p->mostrarInfo();
return 0;
```

Downcasting: conversión de clase base a la clase derivada. Es explícito.

```
Persona * p = new Alumno(555555);
Alumno * a = (Alumno *) p;
a->mostrarInfo();
```

Alumno es subclase de Persona

dynamic_cast

- Verifica que una referencia a un objeto sea de un tipo o de un subtipo dado.
- Podemos utilizarlo antes de convertir un objeto para evitar errores

```
Persona * p = new Alumno(555555);
Alumno * a;
if (a = dynamic_cast<Alumno *>(p)) {
    a->mostrarInfo();
}
```

Alumno no es subclase de Profesor

Métodos y clases *abstract*

El calificador *abstract* condiciona el diseño de una jerarquía de herencia.

Clases Abstract:

- No pueden ser instanciadas.

Métodos Abstract:

- No pueden tener implementación.
- Deben ser implementados para las clases no abstractas que extiendan de su clase.

```
class Persona                          Δ 'Pers
{
private:
    long dni;
    char * nombre;
public:
    Persona(long dni);
    long getDni();
    char * getNombre();
    void setNombre(char * nombre);
    void virtual mostrarInfo();
    void virtual trabajar() = 0;
};
```

Métodos y clases *abstract*

- Cuando un método está presente en todas las clases de una jerarquía pero se implementa en forma diferente conviene definirlo como abstracto.

- Una clase que extiende de una clase abstracta debe:

  - ✔ Implementar todos sus métodos abstractos o …

  - ✔ Declararse como abstracta.

```
class Alumno : public Persona
{
public:
    Alumno(long dni);
    void mostrarInfo();
    void trabajar();
};

#endif // ALUMNO_H
```

```
void Alumno::trabajar() {
    std::cout << "Estudiar !!" << " ";
}
```

Polimorfismo

- Una variable de referencia cambia el comportamiento según el tipo de objeto al que apunta.

- Una variable traba a objetos de una clase como objetos de una superclase y se invoca dinámicamente el método correspondiente (*binding dinámico*)

- Si tenemos un método que espera como parámetro una variable de clase X, podemos invocarlo usando subclases pasando como parámetros referencias a objetos instancia de subclases de X.