HERENCIA

Definición de Herencia

La herencia es el mecanismo que tienen algunos lenguajes como C++ para implementar las relaciones de **especialización** y **generalización**. En este tipo de relación distinguimos la **clase base** (superclase) y la **clase derivada** (subclase). Una clase derivada no puede suprimir atributos o métodos de la clase base; tan solo puede **reescribirlos** (redefinición) o **extenderlos** (debe hacerlo, ya que, si no, no tendría sentido modelar este tipo de relación).

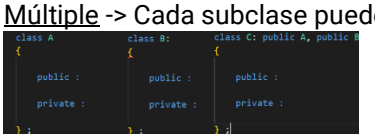
```
class Base
{

    public :

    private :

} ;
```

```
class Derivada: public Base
{

    public :

    private :

}
```

La herencia puede ser:

- Simple -> Cada subclase hereda de una única superclase.

- Múltiple -> Cada subclase puede heredar de múltiples superclases.



Una clase sin clase base se denomina **clase raíz**.

Una clase sin derivadas se denomina **clase hoja.**

Podemos **modelar siguiendo diferentes criterios** (esto es algo que se ve únicamente en el diagrama de relación, pero no en la implementación). Por ejemplo, podríamos hacer que, de una clase base Profesor, partan 2 grupos de subclases: uno para especializar por categoría y otro para especializar por dedicación, pero luego en la implementación se realizaría la herencia como en los ejemplos superiores. Por ejemplo:

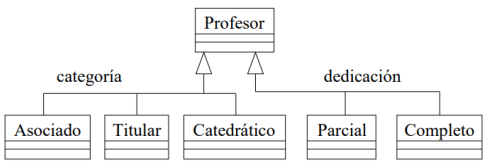


Figura 4.25: Ejemplo de generalización según criterios independientes

```
class Profesor
{

    public :

    private :

} ;
```

```
// Especializamos en función de la categoría
class ProfesorAsociado: public Profesor
{

    public :

    private :

} ;
```

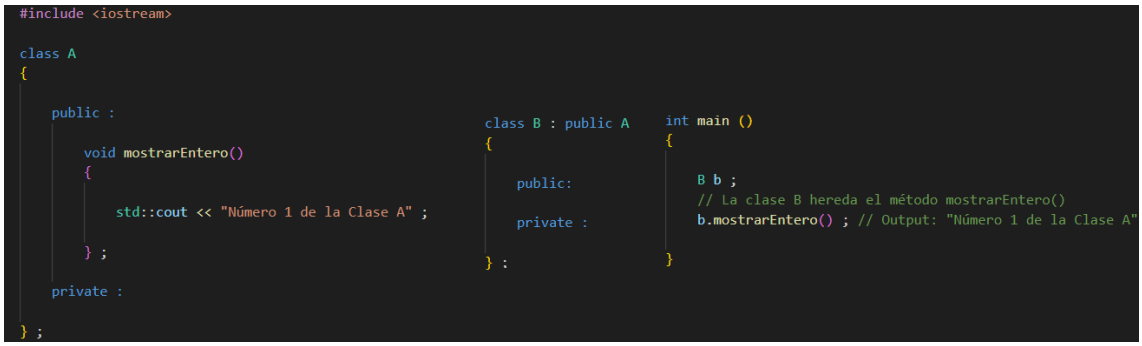
```
// Especializamos en función de la dedicación
class ProfesorTiempoCompleto: public Profesor
{

    public :

    private :

} ;
```

¿Qué ocurre al heredar?

Al realizar una herencia, lo que hacemos es decir que la clase derivada es una subclase de la clase base; es decir, **la clase derivada es un subtipo de la clase** base, por lo que, debería funcionar en todos los casos en los que funcione la clase base. Por esto, siempre que pensemos en modelar una herencia, nos conviene hacernos estas dos preguntas:

> ¿Es la clase derivada una clase base, o es realmente un subtipo de la clase base?
>
> ¿Es posible utilizar la clase derivada en todas las situaciones donde se emplea la clase base?

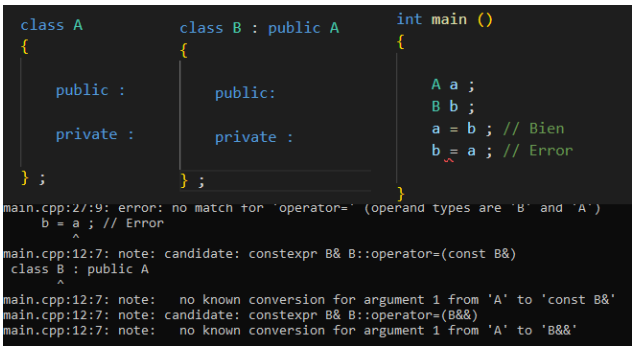
Por ser un subtipo de la clase base, cuando una clase hereda de otra hereda **todos los miembros de la clase base MENOS los constructores** (constructor predeterminado, constructor de copia, constructor de conversión…), destructores y operadores de asignación.



```cpp
#include <iostream>

class A
{

    public :

        void mostrarEntero()
        {

            std::cout << "Número 1 de la Clase A" ;

        } ;

    private :

} ;

class B : public A
{

    public:

    private :

} ;

int main ()
{

    B b ;
    // La clase B hereda el método mostrarEntero()
    b.mostrarEntero() ; // Output: "Número 1 de la Clase A"

}
```

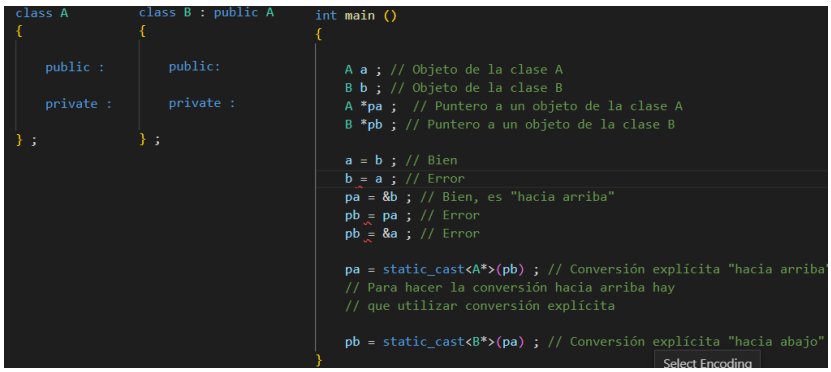
Al ser la clase derivada un subtipo de la clase base, podemos **inicializar objetos de la clase base con objetos de la clase derivada** (perdiéndose los atributos sobrantes); sin embargo, **a la inversa no funciona** ya que habría que utilizar conversión explícita, y esto es inseguro.

Es decir, si tenemos una clase A que hereda de B podríamos hacer A* = B*, pero no B* = A*.



```cpp
class A
{

    public :

    private :

} ;

class B : public A
{

    public:

    private :

} ;

int main ()
{

    A a ;
    B b ;
    a = b ; // Bien
    b = a ; // Error
}
```

```
main.cpp:27:9: error: no match for `operator=` (operand types are `B` and `A`)
     b = a ; // Error
         ^
main.cpp:12:7: note: candidate: constexpr B& B::operator=(const B&)
 class B : public A
       ^
main.cpp:12:7: note:   no known conversion for argument 1 from 'A' to 'const B&'
main.cpp:12:7: note: candidate: constexpr B& B::operator=(B&&)
main.cpp:12:7: note:   no known conversion for argument 1 from 'A' to 'B&&'
```



```cpp
class A
{

    public :

    private :

} ;

class B : public A
{

    public:

    private :

} ;

int main ()
{

    A a ; // Objeto de la clase A
    B b ; // Objeto de la clase B
    A *pa ;  // Puntero a un objeto de la clase A
    B *pb ; // Puntero a un objeto de la clase B

    a = b ; // Bien
    b = a ; // Error
    pa = &b ; // Bien, es "hacia arriba"
    pb = pa ; // Error
    pb = &a ; // Error

    pa = static_cast<A*>(pb) ; // Conversión explícita "hacia arriba"
    // Para hacer la conversión hacia arriba hay
    // que utilizar conversión explícita

    pb = static_cast<B*>(pa) ; // Conversión explícita "hacia abajo"
}
```

Como una clase derivada es un subtipo de la clase base, cuando inicializamos un objeto de la clase derivada, **estamos inicializando un subtipo de la clase base**, por lo que, el constructor de la clase derivada tendrá, como mínimo, los parámetros necesarios para construir un objeto de la clase base, e incluso podemos utilizarlo para construir la clase derivada.

```
class A
{

    public :

        A(int numero):numero_(numero) {}

    private :

        int numero_ ;

} ;
```

```
class B : public A
{

    public:

        B(int numero , bool binario):A(numero),binario_(binario){}

    private :

        bool binario_ ;

} ;
```

A la hora de determinar el tipo de herencia, también es importante el ámbito de los miembros y el ámbito de la herencia (*public, private* o *protected*).

Ámbito

Cuando hablamos de ámbito, podemos distinguir 3 diferentes en función de quién puede acceder o no a la utilización del método/atributo:

- Public: Cualquiera (clases derivadas, clases amigas y clases externas) pueden acceder.
- Protected: La clase derivada y las clases amigas pueden acceder, pero no las clases externas.
- Private: Tan solo pueden acceder los métodos de la propia clase y las clases amigas.

Por omisión, se hereda **privadamente**; no obstante, podemos modificar el ámbito a la hora de declarar la herencia, modificando de esta manera el acceso a los miembros:

Accesibilidad	un miembro de la clase base...	pasa a ser...
`public`	público	público
	protegido	protegido
	privado	inaccesible
`protected`	público	protegido
	protegido	protegido
	privado	inaccesible
`private`	público	privado
	protegido	privado
	privado	inaccesible

```
class Base {
public:
  int publico;
protected:
  int privado;
private:
  int protegido;
};

class DerivadaPublica: public Base {
  // ...
};

class DerivadaProtegida: protected Base {
  // ...
};

class DerivadaPrivada: private Base {
  // ...
};
```

En *DerivadaPublica*, *publico* seguirá siendo público y *protegido* seguirá siendo protegido. O sea, al heredar con `public` se conserva la visibilidad de los miembros públicos y protegidos.

En *DerivadaProtegida*, los miembros *publico* y *protegido* heredados pasan a ser protegidos.

Por último, en la clase *DerivadaPrivada*, los miembros *publico* y *protegido* heredados serán ahora miembros privados.

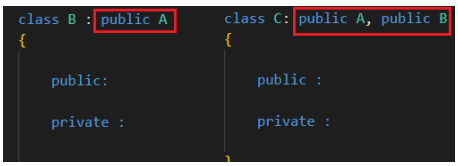
En los tres casos el miembro *privado* de la clase base, que es heredado por las clases derivadas, será inaccesible. Esto es así porque un miembro privado es invisible desde el exterior de la clase a la que pertenece, incluso para sus derivadas. Este miembro privado sólo podrá ser manejado a través de las funciones públicas que la clase base haya dispuesto para ello.

Orden de Inicialización

Cuando una clase hereda de otra (herencia simple), no hay dudas respecto al orden de inicialización: como construimos un objeto de una clase derivada, primero se instancia la clase base, después los atributos correspondientes, y por último la propia clase.

Con herencia simple es sencillo, pero a la hora de la herencia múltiple pueden surgirnos dudas, por lo que es importante tener siempre claro el orden de inicialización:

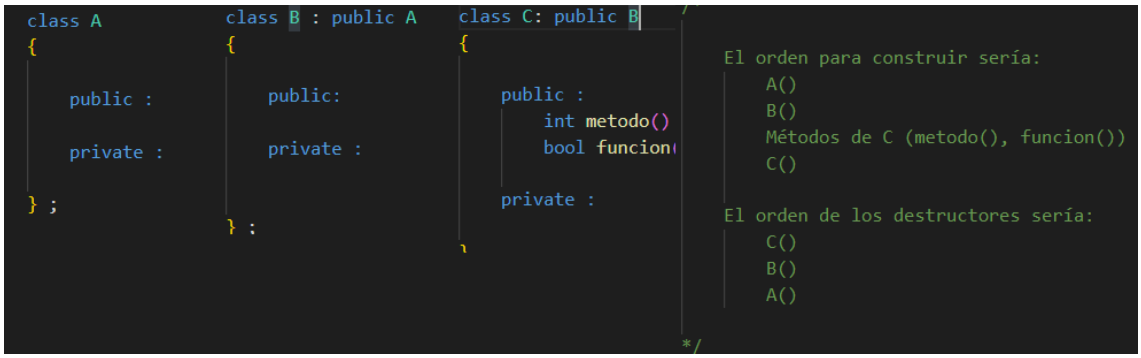
1. Ver el orden de instanciación de clases según la **lista de herencia.**



2. Instanciar las clases bases.
3. Instanciar los atributos de la clase derivada.
4. Instanciar el objeto de la clase derivada.

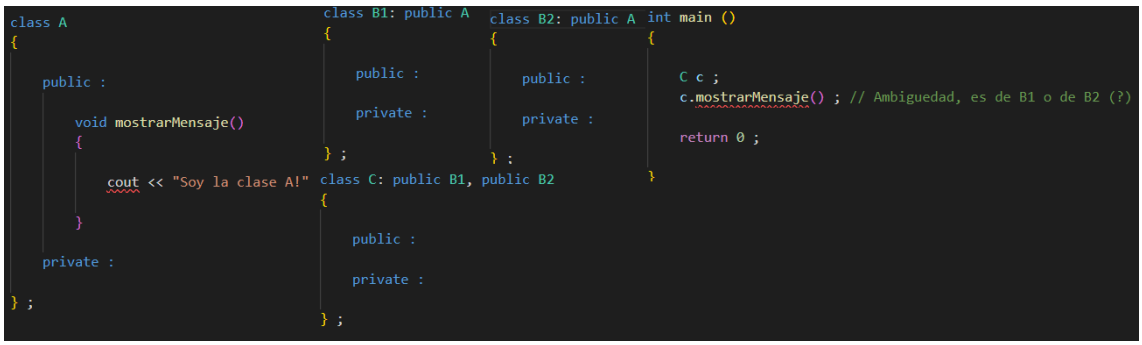
Es decir, si D hereda de B y C, y estos a su vez heredan de una clase A, primero se inicializa el objeto de A, después B y C (según el orden en el que aparezcan en la lista de herencia), los miembros de D, y por último el objeto de D.

Los destructores actúan en orden inverso



Problemas con la Herencia Duplicada

Cuando varias clases heredan de la misma nos podemos encontrar con problemas para resolver la ambigüedad de métodos o atributos. Por ejemplo, imaginemos que teneos una clase D que hereda de B y C, y estas a su vez (B y C), heredan de A que tiene un método f(). La clase D habrá heredado B::f() por parte de la herencia de B con A, y C::f() por parte de la herencia de C con A, por lo que si llamamos a este método a través de un objeto de D sin especificar el ámbito se producirá un error, o sea, si hacemos d.f(). Veámoslo con un ejemplo:



Para solventar esta ambigüedad tendremos que resolver el ámbito:

```
class A
{

    public :

        void mostrarMensaje()
        {

            cout << "Soy la clase A!"

        }

    private :

} ;
```

```
class B1: public A
{

    public :

    private :

} ;
class C: public B1, public B2
{

    public :

    private :

} ;
```

```
class B2: public A
{

    public :

    private :

} ;
```

```
int main ()
{

    C c ;
    c.B1::mostrarMensaje() ; // Resolvemos el ámbito
    c.B2::mostrarMensaje() ; // Resolvemos el ámbito

    return 0 ;

}
```

O, debido a que, si un mismo método o atributo se hereda por caminos diferentes, lo normal es que tan solo se requiera una única copia de ese elemento, y no dos, por lo que podemos utilizar la palabra reservada **virtual**. Esta es una palabra reservada que se aplica a declaraciones para conseguir un **polimorfismo en tiempo de ejecución**.

```
class A
{

    public :

        void mostrarMensaje()
        {

            cout << "Soy la clase A!" ;

        }

    private :

} ;
```

```
class B1: virtual public A
{

    public :

    private :

} ;
class C: public B1, public B2
{

    public :

    private :

} ;
```

```
class B2: virtual public A
{

    public :

    private :

} ;
```

```
int main ()
{

    C c ;
    c.mostrarMensaje() ; // Graicas al virtual no existe ambiguedad

    return 0 ;

}
```

Aunque también podemos tener situaciones en la que, sobrecarguemos los métodos de la clase base (hacemos una redefinición) en la clase derivada. Ante esto también es conveniente resolver el ámbito para evitar que se produzcan errores de ambigüedad por conversiones implícitas. Por ejemplo:

```
#include <iostream>
using namespace std;

class B1 {
public:
  void f(int i) { cout << "B1::f(int)" << endl; }
  // ...
};

class B2 {
public:
  void f(double d) { cout << "B2::f(double)" << endl; }
  // ...
};

class D: public B1, public B2 {
  // ...
};
```

```
int main()
{
  D d;
  d.f(0);       // ERROR, ¿qué f(), el de B1 o el de B2?
  d.f(0.0);     // ERROR, ¿qué f(), el de B1 o el de B2?
  d.B1::f(0);   // bien, B1::f()
  d.B2::f(0.0); // bien, B2::f()
}
```

```
#include <iostream>

class B1 {
public:
  void f(char i) { std::cout << "B1::f(char)" << std::endl; }
  // ...
};

class B2 {
public:
  void f(int d) { std::cout << "B2::f(int)" << std::endl; }
  // ...
};

class D: public B1, public B2 {
public:
  using B1::f; using B2::f;
  void f(double c) { std::cout << "D::f(double)" << std::endl; }
  // ...
};
```

```
int main()
{
  D d;
  d.f('A');  // B1::f(char)
  d.f(0);    // B2::f(int)
  d.f(0.0);  // D::f(double)
}
```

Overshadow

Si tenemos una clase derivada con un método que se llama igual que el de la clase base, el de la clase base quedará oculto a menos que resolvamos el ámbito explícitamente.

```
class A
{

    public :

        void mostrarMensaje()
        {

            std::cout << "Soy la clase A!" ;

        }

    private :

} ;
```

```
class B: public A
{

    public :

        void mostrarMensaje() // Overshadow
        {

            std::cout << "Soy de la clase B!" ;

        }

} ;
```

```
int main ()
{

    B b ;
    b.mostrarMensaje() ; // "Soy de la clase B!"
    b.A::mostrarMensaje() ; // "Soy de la clase A!"

}
```

Agregación/Composición VS Herencia

Por la naturaleza de la especialización, en ocasiones puede confundirse con una agregación; sin embargo, no debe confundirse (de hecho, esta es una pregunta teórica que suelen meter en los exámenes).

En una **agregación** participan **2 clases diferentes que tienen relación entre ellas**, lo que no implica que una sea un subtipo de otra. En cambio, en una especialización, estamos creando un subtipo de una clase base.

Por ejemplo, si nos dicen que se quiere implementar una clase Pila y ya se dispone de una clase Lista, lo correcto sería modelarla como una agregación/composición, ya que, realmente no podemos utilizar una Pila en todos los casos que utilizamos una Lista (si alguna vez tienes dudas, hazte las dos preguntas que encontrarás al inicio de este mismo documento). Además, si implementásemos una Pila como clase derivada de una clase Lista, la clase Pila heredaría miembros de la clase Lista que no tendría sentido utilizar sobre una Pila, y que, de ser utilizados, harían que la Pila no se comportase como tal. Por este motivo, lo correcto sería definir la clase Pila como una agregación/composición de la clase Lista, y delegar el comportamiento de la clase Pila en las operaciones de la clase Lista que nos interesen.