FACULTAD DE INFORMÁTICA

Herencia

TEMA

Programación orientada a objetos — Unidad 6

Autor: Luis Hernández Yáñez

FdI UCM

Contenido

FdI UCM

Contenido

FdI UCM

Introducción

Mecanismo exclusivo y fundamental de la POO.

La herencia no está contemplada en la programación basada en tipos (TAD).

Es el principal mecanismo que ayuda a fomentar y facilitar la reutilización del software:
Las clases como componentes software reutilizables.

Si se necesita una nueva clase de objetos y se detectan *suficientes* similitudes con otra clase ya desarrollada, se toma esa clase existente como punto de partida para desarrollar la nueva:

- ✓ Se adoptan automáticamente características ya implementadas
  ⇒ Ahorro de tiempo y esfuerzo
- ✓ Se adoptan automáticamente características ya probadas
  ⇒ Menor tiempo de prueba y depuración

Se puede partir de varias clases (herencia múltiple).

Introducción

Base de la aplicación del mecanismo de herencia:

- ✓ Suficientes similitudes:
  Todas las características de la clase existente
  (o la gran mayoría de ellas) resultan adecuadas para la nueva.
- ✓ En la nueva clase se ampliará y/o redefinirá
  el conjunto de características.

Características de las clases que se adoptan:
Todos los miembros definidos en las clases.

- ✓ Atributos
- ✓ Métodos (funciones miembro)

Dependiendo de la forma en que se aplique el mecanismo de herencia, en la nueva clase se puede tener o no acceso a ciertas características heredadas (a pesar de que se adopten todas).

Terminología

La relación de herencia se establece entre una nueva clase (referida aquí con el nombre **Nueva**) y una clase ya existente (referida aquí con el nombre **Existente**).

Un poco de terminología:

- ✓ **Existente** se dice que es la clase base (término de C++),
  la clase madre o la superclase (término genérico de la POO).
- ✓ **Nueva** se dice que es la clase derivada (término de C++),
  la clase hija o la subclase (término genérico de la POO).
- ✓ En C++ también se utiliza el término derivación para referirse a la herencia.
- ✓ La clase **Nueva** es la que tiene establecida la relación de herencia con la clase **Existente**; **Existente** no necesita a **Nueva**, pero **Nueva** sí necesita la presencia de **Existente**.

La nueva clase que hereda

La relación de herencia se establece entre una clase **Nueva** y una clase **Existente**.

Sobre la clase que hereda de la existente:

- ✓ **Nueva** hereda todas las características de **Existente**.
- ✓ **Nueva** puede definir características adicionales.
- ✓ **Nueva** puede *redefinir* características heredadas de **Existente**.
- ✓ **Nueva** puede anular características heredadas de **Existente**.
- ✓ El proceso de herencia no afecta de ninguna forma a la superclase **Existente**.

Especialización frente a generalización

Cuando se crea una subclase a partir de una clase existente (superclase), en principio se puede hacer que la subclase sea más general que la superclase o que la subclase sea más específica que la superclase.

Sin embargo, el enfoque habitual (y más útil en general) es considerar las clases como tipos y la herencia como un mecanismo de creación de subtipos.

Así, los lenguajes de programación basados en tipos (los más utilizados) interpretan la herencia como una especialización y, por tanto, proporcionan un mecanismo de herencia adecuado y cómodo para la especialización, pero que resulta difícil de aplicar para crear subclases más generales que sus superclases.

C++ no es una excepción, por lo que aplicamos la herencia como un mecanismo de especialización: las subclases son más específicas que las superclases (refinamiento / menor nivel de abstracción).

Herencia en C++: la superclase

La superclase (clase base) podrá ser cualquier clase existente (ya desarrollada).

Recordemos los tipos de miembros y sus posibles accesos:

- ✓ Miembros privados: atributos (y puede que algunos métodos *internos*).
- ✓ Miembros públicos: métodos que implementan los servicios que se proporcionan *al exterior*.
- ✓ Tanto los miembros públicos como los privados son accesibles en cualquier lugar de la implementación de la clase (en todas las funciones miembro, públicas o privadas).
- ✓ Sólo los miembros públicos son accesibles desde fuera de la clase (mediante pasos de mensajes a ejemplares de la clase).

Esto no es nada nuevo, ya que se aplica a cualquier clase, independientemente de que sea superclase de otra(s) o no.

Herencia en C++: la subclase

La subclase (clase derivada) se crea indicando explícitamente la (super)clase de la que se parte (hereda o deriva)*. Esto se hace colocando dos puntos (:) y el nombre de la superclase entre el nombre de la subclase y la llave de apertura de la estructura **class**.

```
class Subclase : Superclase {
```

- ✓ Se adoptan automáticamente todos los miembros públicos y privados de la superclase.
- ✓ Se pueden definir miembros propios adicionales (públicos y privados), así como redefinir miembros heredados.
- ✓ Importante: en la implementación de la subclase (en las funciones miembro definidas en la propia subclase) *no se tiene acceso a lo privado de la superclase*, a pesar de que se trate de miembros que se heredan de la misma forma que los públicos.

* Habrá varias superclases cuando se aplique herencia múltiple [*Suplemento*].

Herencia en C++: modos de derivación

C++ contempla distintos modos de derivación (aplicación del mecanismo de herencia). Dependiendo del modo de derivación que se aplique, el acceso (público/privado) de los miembros heredados se mantiene o se restringe en la subclase.

Al crear una subclase, el modo de derivación a aplicar se indica con una palabra reservada precediendo el nombre de la superclase.

Podemos aplicar dos modos de derivación*:

- ✓ Derivación pública: en la subclase cada miembro heredado mantiene el modo de acceso establecido en la superclase.
- ✓ Derivación privada: todos los miembros heredados (públicos y privados) se convierten en miembros privados en la subclase.

Por defecto (si no se indica explícitamente un modo de derivación), se aplica el modo de derivación privado.

* Se puede aplicar un tercer modo de derivación que se verá más adelante.

Herencia en C++: modo de derivación público

Herencia con modo de derivación público:

Palabra reservada **public** precediendo el nombre de la superclase.

Los miembros heredados mantienen su modo de acceso:

- ✓ Los que son públicos en la superclase siguen siendo públicos (y accesibles) en la subclase.
- ✓ Los que son privados en la superclase siguen siendo privados en la subclase y se ocultan, resultando inaccesibles en ésta.

En las funciones miembro de la subclase no se tiene acceso a los miembros privados heredados (esto siempre es así).

A los objetos de la subclase se les puede seguir pasando mensajes que correspondan a métodos públicos heredados.

Es, con diferencia, el modo de derivación más habitual.

```
class Subclase : public Superclase { ...
```

Herencia en C++: modo de derivación privado

Herencia con modo de derivación privado (el predeterminado):

Palabra reservada **private** precediendo el nombre de la superclase.

Todos los miembros heredados se convierten en privados:

- ✓ Los que son públicos en la superclase pasan a ser privados en la subclase, pero no se ocultan y se pueden acceder en ésta.
- ✓ Los que son privados en la superclase siguen siendo privados en la subclase y se ocultan, resultando inaccesibles en ésta.

En las funciones miembro de la subclase no se tiene acceso a los miembros privados heredados, pero sí a los miembros públicos heredados, aunque se hayan convertido en privados.

A los objetos de la subclase no se les puede pasar mensajes que correspondan a métodos públicos heredados (se han convertido en privados en la subclase).

```
class Subclase : private Superclase { ...
```


Herencia en C++: un ejemplo

Recordemos la clase **Persona** de unidades anteriores:

```
class Persona {
public: ...
  // miembros públicos
private: ...
  // miembros privados
};
```

Supongamos que queremos implementar una clase **Alumno**:

- ✓ Los alumnos también tienen NIF, nombre, apellidos y edad: Los atributos de la clase **Persona** son adecuados para **Alumno**.
- ✓ Los métodos de la clase **Persona**, en principio, parecen todos ellos adecuados para la clase **Alumno**.

*Los alumnos **son** personas*

Resulta adecuado aplicar el mecanismo de herencia y crear la nueva clase (**Alumno**) a partir de la existente (**Persona**).

Herencia en C++: un ejemplo

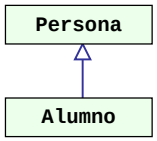
```
class Persona {
public: ...
  // miembros públicos
private: ...
  // miembros privados
};

class Alumno : public Persona {
...
```

Superclase (clase base)

Modo de derivación público

Subclase (clase derivada)

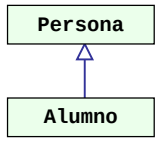


Sin tener que hacerlo explícito (sin tener que volver a declararlos):

- ✓ La clase **Alumno** incorpora los miembros privados de **Persona** Los miembros privados heredados siguen siendo privados.
- ✓ La clase **Alumno** incorpora los miembros públicos de **Persona** Los miembros públicos heredados siguen siendo públicos.

Sin embargo, las funciones miembro que se definan en la clase **Alumno** no podrán acceder a los miembros privados heredados.

class Alumno : public Persona { ...

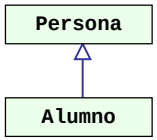



```
clase Persona
privado: _nif  _nombre  _apellidos  _edad
público: nif(string)   nombre(string)
         apellidos(string)   edad(int)
         nif()   nombre()   apellidos()
         edad()   leer()   mostrar()
         nombreCompleto()   felizCumple()

clase Alumno: características heredadas
privado: _nif  _nombre  _apellidos  _edad   (y no accesibles)
público: nif(string)   nombre(string)
         apellidos(string)   edad(int)
         nif()   nombre()   apellidos()
         edad()   leer()   mostrar()
         nombreCompleto()   felizCumple()
```

Más adelante veremos cómo afecta la herencia a los constructores, destructores y operadores de asignación.

Herencia en C++: un ejemplo



Por defecto el modo de derivación es privado.

Esto significa que

```
class Alumno : Persona { ...
```

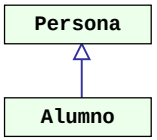
es equivalente a

```
class Alumno : private Persona { ...
```

Para evitar confusiones y facilitar la comprensión del código, siempre indicaremos explícitamente el modo de derivación empleado, precediendo el nombre de la superclase con la palabra clave **public** o la palabra clave **private**.

No poner la palabra clave correspondiente hace que se deba recordar cuál es el valor por defecto, y cuanta menos información predeterminada se deba recordar, más fácil resultará leer el código.

class Alumno : private Persona { ...




```
clase Persona
privado: _nif  _nombre  _apellidos  _edad
público: nif(string)   nombre(string)
         apellidos(string)   edad(int)
         nif()   nombre()   apellidos()
         edad()   leer()   mostrar()
         nombreCompleto()   felizCumple()

clase Alumno: características heredadas
privado: _nif  _nombre  _apellidos  _edad    (no accesibles)
         nif(string)   nombre(string)
         apellidos(string)   edad(int)
         nif()   nombre()   apellidos()      (accesibles)
         edad()   leer()   mostrar()
         nombreCompleto()   felizCumple()
público: nada
```

Más adelante veremos cómo afecta la herencia a los constructores, destructores y operadores de asignación.

Modo de derivación público: acceso a lo heredado

Funciones miembro de la subclase:
No tienen acceso a las características privadas heredadas.

```
clase Persona
privado:
   _nif        _nombre
   _apellidos  _edad
público:
   nif(string)   nombre(string)
   apellidos(string)   edad(int)
   nif()   nombre()   apellidos()
   edad()   leer()   mostrar()
   nombreCompleto()
   felizCumple()

class Alumno : public Persona {
...
```

Privado en **Alumno**. Inaccesible en sus funciones miembro. Inaccesible desde fuera de los objetos.

Público en **Alumno**. Accesible en sus funciones miembro. Accesible desde fuera de los objetos (pasos de mensajes).

Modo de derivación privado: acceso a lo heredado

Funciones miembro de la subclase:
No tienen acceso a las características privadas heredadas.

```
clase Persona
privado:
   _nif        _nombre
   _apellidos  _edad
público:
   nif(string)   nombre(string)
   apellidos(string)   edad(int)
   nif()   nombre()   apellidos()
   edad()   leer()   mostrar()
   nombreCompleto()
   felizCumple()

class Alumno : private Persona {
...
```

Privado en **Alumno**. Inaccesible en sus funciones miembro. Inaccesible desde fuera de los objetos.

Privado en **Alumno**. Accesible en sus funciones miembro. Inaccesible desde fuera de los objetos.

¿Necesidad de acceso a miembros privados heredados?

A menudo resultará necesario disponer en la subclase de algún medio de acceso a los miembros privados heredados.

Por ejemplo, si en la clase **Alumno** un método necesita conocer la edad para tomar determinadas decisiones, será necesario proporcionar al método algún medio de acceso al atributo **_edad**, atributo heredado de la superclase **Persona**, donde es privado, lo que impide su acceso directo en la subclase **Alumno**.

La superclase **Persona** tiene definidos accedentes y mutadores públicos para sus atributos, por lo que en la subclase **Alumno** se puede acceder a los atributos a través de ellos.

Sin embargo, hay ocasiones en las que se puede necesitar (o simplemente desear) un acceso directo en la subclase a los miembros privados heredados de la superclase.

¿Necesidad de acceso a miembros privados heredados?

Motivos por los que puede resultar conveniente proporcionar un acceso directo a los miembros privados heredados:

- ✓ No resulta procedente proporcionar en la superclase accedentes y/o mutadores para determinados atributos (y se necesita accederlos en la subclase).
- ✓ Hay miembros privados que son métodos auxiliares de la superclase y que, por tanto, no se proporcionan *al exterior* como servicios (puede resultar necesario su uso en la subclase).
- ✓ Simplemente por comodidad: la implementación resulta algo más sencilla si se accede directamente a los atributos, en lugar de accederlos por medio de accedentes y mutadores. (El motivo más habitual de los programadores.)

Cualquiera que sea el motivo, la única forma que conocemos de proporcionar en la subclase un acceso directo a los miembros privados heredados es hacer que sean públicos en la superclase.

Otro modo de acceso: miembros protegidos

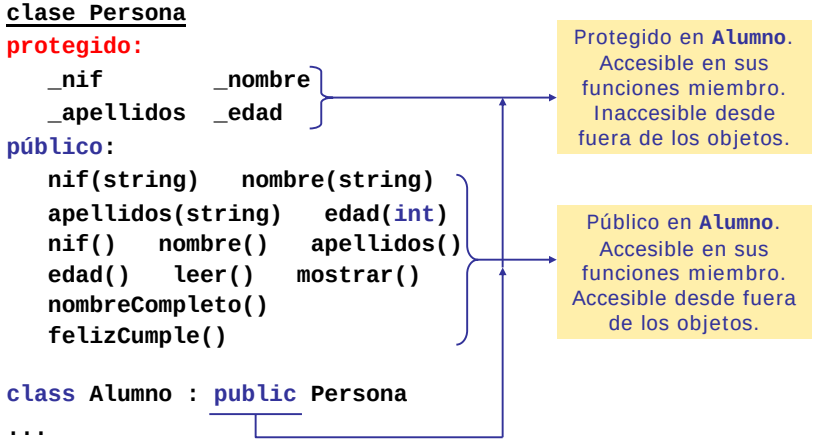
Por ejemplo, para que el atributo **_edad** se pueda acceder directamente en las funciones miembro de la subclase (**Alumno**), deberá estar declarado como público en la superclase (**Persona**).

Pero, ¡¡¡eso no sería correcto!!! (¿un atributo público?)

Solución: establecer el modo de acceso de los miembros privados de la superclase como protegido.

- ✓ Los miembros protegidos se definen en una sección **protected** de la estructura **class**.
- ✓ Los miembros protegidos se comportan en la clase en la que están definidos como si fueran privados.
- ✓ Las funciones miembro de una subclase sí tienen acceso a los miembros protegidos heredados de la superclase.
- ✓ En una derivación pública, los miembros protegidos heredados siguen siendo protegidos en la subclase.

Miembros protegidos (modo de derivación público)



Miembros protegidos (modo de derivación privado)

```
clase Persona
protegido:
   _nif        _nombre
   _apellidos  _edad
público:
   nif(string)   nombre(string)
   apellidos(string)   edad(int)
   nif()   nombre()   apellidos()
   edad()   leer()   mostrar()
   nombreCompleto()
   felizCumple()

class Alumno : private Persona
...
```

Protegido en **Alumno**. Accesible en sus funciones miembro. Inaccesible desde fuera de los objetos.

Privado en **Alumno**. Accesible en sus funciones miembro. Inaccesible desde fuera de los objetos.

Herencia en C++: accesibilidad en la derivación

En este curso no usaremos miembros protegidos. Los atributos siempre irán en la sección **private**.

	Miembros de la superclase		
	públicos	protegidos	Privados
En la subclase			
Con derivación pública:	públicos	protegidos	privados (ocultos)
Accesibles a las funciones miembro	SI	SI	NO
Accesibles desde fuera de los objetos	SI	NO	NO
Con derivación privada:	privados	protegidos	privados (ocultos)
Accesibles a las funciones miembro	SI	SI	NO
Accesibles desde fuera de los objetos	NO	NO	NO

Miembros que no se heredan

No todos los miembros de la superclase se heredan en la subclase.

Los constructores (predeterminado y de copia), el destructor y el operador de asignación no se heredan en las subclases.

Sin embargo, si en la subclase no están definidos, el compilador añade automáticamente unos que resulten adecuados.

Constructores y destructores:

- ✓ Si en la subclase no están definidos, se crean automáticamente un constructor predeterminado y un destructor, aunque sin que hagan nada concreto (se necesitan porque serán invocados).
- ✓ Cuando se crea un ejemplar (objeto) de la subclase, se invocan automáticamente todos los constructores de todas las superclases, en el mismo orden establecido por la jerarquía (de la más general a la más específica), y finalmente el constructor propio de la subclase.

Miembros que no se heredan

- ✓ Cuando se destruye un ejemplar (objeto) de la subclase, se invocan automáticamente todos los destructores de todas las superclases, en orden inverso al establecido por la jerarquía (de la más específica a la más general), comenzando por el destructor propio de la subclase.

Se deben crear explícitamente los constructores y el destructor en la subclase (*ajustarse a la Forma canónica ortodoxa*).

Copia (operador de asignación):

- ✓ Si en la subclase no está definido, se crea automáticamente un operador de asignación que se basa en el que está definido en la superclase (copia todo lo heredado como ella y bit a bit lo propio).

Se debe crear explícitamente un operador de asignación para la copia en la subclase (*ajustarse a la Forma canónica ortodoxa*).

Implementación de la subclase **Alumno**

Una vez decidido que la clase **Alumno** debe ser subclase de **Persona**, el proceso de desarrollo de esa nueva clase (**Alumno**) prosigue centrándose en las características de ésta que son diferentes de las de la superclase:

- ✓ Atributos adicionales: **_curso** (más tarde añadiremos otro).
- ✓ Métodos adicionales (públicos y privados): accedentes y mutadores para los nuevos atributos.
- ✓ Redefinición de métodos heredados que no sean del todo adecuados para las características específicas de la subclase.

De momento, nos centraremos en los dos primeros puntos y en hacer que la subclase se ajuste a la Forma canónica ortodoxa.

Partimos de la clase **Persona** ya desarrollada, con atributos privados, no protegidos. **protected** sólo debe ser utilizado cuando sea realmente necesario, no sólo por comodidad.

Implementación de la subclase **Alumno**

```
class Alumno : public Persona {
public:
  Alumno(string nif = "", int edad = 0,
    string nombre = "", string apellidos = "",
    int curso = 1)
    : _curso(curso),
    Persona(nif, edad, nombre, apellidos) { }

  Alumno(const Alumno&); //  constructor de copia
  Alumno& operator=(const Alumno&); // copia
  ~Alumno() {} // destructor
  void curso(int num) { _curso = num; } // mutador
  int curso() const { return _curso; } // accedente
private:
  int _curso; // nuevo atributo
};
```

Paso de argumentos al constructor de la superclase

Implementación de la subclase **Alumno**

```
Alumno::Alumno(const Alumno& otro)
{
  nif(otro.nif());
  edad(otro.edad());
  nombre(otro.nombre());
  apellidos(otro.apellidos());
  _curso = otro._curso;
}

Alumno& Alumno::operator=(const Alumno& otro)
{
  nif(otro.nif());
  edad(otro.edad());
  nombre(otro.nombre());
  apellidos(otro.apellidos());
  _curso = otro._curso;
  return *this;
}
```


Creación y destrucción de objetos **Alumno**

Ya tenemos una primera implementación de la subclase **Alumno** y podemos crear ejemplares suyos:

```
int main()
{
  Alumno unAlumno;
  ...
```

Al crear un ejemplar de la clase **Alumno** (subclase de **Persona**), la construcción del objeto se lleva a cabo de la siguiente forma:

1. Se invocan los constructores de las superclases, desde el de la más general hasta el de la inmediata. En este caso la única superclase es **Persona**. La invocación del constructor de **Persona** se realiza con los argumentos que se especifican en la lista de inicializadores de la subclase: **Persona(nif, edad, nombre, apellidos)**. Si no se pasan argumentos en el de la subclase, se usa el constructor predeterminado.

Creación y destrucción de objetos **Alumno**

2. Se ejecuta el constructor de la superclase **Persona**: inicializa los atributos de acuerdo con la lista de inicializadores y luego ejecuta el cuerpo de la función (que no hace nada).
3. Se ejecuta el constructor de la propia (sub)clase **Alumno**: inicializa el atributo **_curso** y luego ejecuta el cuerpo de la función (que tampoco hace nada adicional).

Cuando se destruye un ejemplar de la (sub)clase **Alumno**, de momento cuando termina la ejecución del ámbito en el que está definido (programa, función o bloque), ocurre lo siguiente:

1. Se invoca el destructor de la propia clase **Alumno**, (que en este caso no hace nada).
2. Se invocan los destructores de todas las superclases, comenzando por el de la inmediata y siguiendo hacia arriba en la secuencia de derivación. En este caso la única superclase es **Persona**, por lo que su destructor es el único que se invoca (tampoco hace nada).

Creación de objetos **Alumno** por copia

Veamos qué ocurre cuando se crea un ejemplar de la clase **Alumno** como copia de otro ya existente:

```
int main()
{
  Alumno alumno1;
  ...
  Alumno alumno2(alumno1); ...
```

Como es de esperar, se invoca el constructor de copia de la propia clase **Alumno**. NO se produce ninguna invocación automática de ningún tipo de constructor (ni el de copia) de sus superclases.

El proceso es el mismo cuando se crea la copia de un argumento para un parámetro declarado con paso por valor.

Si no hubiéramos creado un constructor de copia en la subclase, el compilador habría añadido automáticamente uno tomando como base el de la superclase y añadiendo copias bit a bit para los atributos propios de la subclase.

Copia de objetos **Alumno**

Veamos qué ocurre cuando se copia un ejemplar de la clase **Alumno** en otro:

```
int main()
{
  Alumno alumno1, alumno2;
  ...
  alumno2 = alumno1; ...
```

Como es de esperar, se invoca el operador de asignación de la propia clase **Alumno**. NO se produce ninguna invocación automática de ningún operador de asignación de sus superclases.

Si no hubiéramos creado un operador de asignación en la subclase, el compilador habría añadido automáticamente uno tomando como base el de la superclase y añadiendo copias bit a bit para los atributos propios de la subclase.

Más sobre los métodos que no se heredan

Los constructores, destructores y operadores de asignación son métodos especiales que, como ya sabemos, no se heredan.

Como no se heredan, no se pueden ejecutar en la subclase simplemente invocándolos sin poner delante objeto receptor (como podemos hacer con cualquier método heredado accesible).

Sin embargo, puede resultar conveniente invocar esos métodos especiales de la superclase en los correspondientes de la subclase.

Comencemos por el constructor de copia de la subclase **Alumno**: como es de esperar, copia los atributos heredados y los atributos propios. El código que copia los atributos heredados es el mismo que el código del constructor de copia de la superclase **Persona**.

Podemos hacer que la copia de los atributos heredados la realice el constructor de copia de la superclase, forzando su ejecución en el constructor de copia de la subclase. Esto simplificará el código de este último (imagínese que hubiera 40 atributos heredados).

Reutilización del constructor de copia de la superclase

Como el constructor de copia de la superclase no se hereda en la subclase, se debe invocar explícitamente. El nombre del constructor de copia es el mismo que el nombre de la clase.

La invocación se realiza a modo de incializador, a continuación de la cabecera de la función constructora de copia de la subclase:

```
Alumno::Alumno(const Alumno& otro) : Persona(otro) {
  _curso = otro._curso;
}
```

Más adelante veremos que se puede pasar un objeto de la subclase (**otro**) como argumento de un parámetro de la superclase.

Reutilización del operador de copia de la superclase

¿Qué ocurre con el operador de asignación?

Como es de esperar, también copia los atributos heredados y los atributos propios. El código que copia los atributos heredados es el mismo que el código del operador de asignación de la superclase.

Podemos hacer que la copia de los atributos heredados la realice el operador de asignación de la superclase, forzando su ejecución en el de la subclase. Esto simplificará el código de este último.

El operador de asignación tampoco se hereda, pero se puede invocar explícitamente dentro de la función operadora de la subclase:

```
Alumno& Alumno::operator=(const Alumno& otro) {
  Persona::operator=(otro);
  _curso = otro._curso;
  return *this;
}
```


Servicios proporcionados por **Persona**s y **Alumno**s

Personas	**Alumnos**
nif(string)	**nif(string)**
nombre(string)	**nombre(string)**
apellidos(string)	**apellidos(string)**
edad(int)	**edad(int)**
nif()	**nif()**
nombre()	**nombre()**
apellidos()	**apellidos()**
edad()	**edad()**
nombreCompleto()	**nombreCompleto()**
felizCumple()	**felizCumple()**
leer()	**leer()**
mostrar()	**mostrar()**
	curso(int)
	curso()

Heredados

Aparte de constructores, destructor y operador de asignación

Redefinición de métodos

En la subclase **Alumno** hemos ampliado el conjunto de métodos heredado, definiendo nuevos métodos en la subclase.

Frecuentemente, algún método de la superclase no resultará (del todo) adecuado para la subclase.

Se puede *redefinir* el método en la subclase sin más que declarar una función miembro con el mismo nombre.

El método **mostrar()** de la superclase **Persona** no resulta del todo adecuado para los objetos de la subclase **Alumno**, ya que no muestra la información correspondiente al atributo propio **_curso**:

```
void Persona::mostrar() const {
  cout << nif() << endl;
  cout << nombreCompleto() << endl;
  cout << "Edad: " << edad() << endl;
}
```

Redefinición de métodos

Redefinimos el método **mostrar()** heredado definiendo en la subclase **Alumno** una función miembro con ese mismo nombre:

```
class Alumno : public Persona {
public:
...
  void mostrar() const; // método redefinido
...
};
```

Ahora tenemos definido un método **mostrar()** propio en la subclase y será éste el que se ejecute cuando se pasen mensajes **mostrar()** a los objetos de la subclase **Alumno**.

Redefinición de métodos

El método redefinido en la subclase se implementa con el código adecuado para los objetos de la subclase.

Si, como en este caso, la no adecuación del método de la superclase se debe a que no hace todo lo que se necesita, pero su código es una parte válida del total de código necesario, se puede aprovechar el código del método de la superclase:

```
void Alumno::mostrar() const
{
  Persona::mostrar();
  cout << "Curso: " << _curso << endl;
}
```

Ejecución del código del método de la superclase

Seguimos el mismo proceso al redefinir el método **leer()**.

La implementación final (de momento) de la subclase **Alumno** se deja como ejercicio (separando cabecera e implementación).