Todo sobre 2º parcial
Programación Orientada a Objetos

Hecho por: Jose Luis Venega Sánchez

Índice general

1. Teoría de la STL

La *STL* (Standard Template Library) de C++ es una parte integral del estándar de C++ que proporciona una colección de plantillas y algoritmos genéricos, así como tipos de datos contenedores y funciones asociadas para facilitar el desarrollo de software.

La *STL* fue diseñada para ser flexible, eficiente y fácil de usar.

En este último caso encontramos `set` (conjunto de objetos de un tipo determinado) y `map` (diccionario clave - valor). Además encontramos sus variantes donde podemos encontrar valores repetidos como son `multiset` ó `multimap` y sus variantes donde el contenido de estos contenedores no están ordenados `unordered_set` y `unordered_map`, respectivamente.

La *STL* proporciona una amplia gama de algoritmos genéricos que operan con los tipos de datos contenedores como: vectores `vector`, listas doblemente enlazadas `list`, bicolas `deque`, pilas `stack` o contenedores que contienen una colección de objetos de un tipo determinado.

1.1. Iteradores

Mediante los `iterators` (iteradores) podemos acceder a las posiciones o elementos contenidos en cada tipo de contenedor. Se recomienda el uso de la *keyword* `auto`, para que el compilador le asigne el tipo adecuado al iterador.

Librería → `<iterator>`

Todos los tipos contendores asociativos pueden hacer uso de estos iteradores, donde encontramos varios métodos de acceso a las posiciones de los contenedores:

- `begin()` (puntero a la primera posición) y `cbegin()`(puntero constante a la primera posición).
- `end()` (puntero a la siguiente posición del final del contendor) y `cend()` (puntero constante a la siguiente posición del final del contendor).

1.2. Contenedores asociativos de la STL

Vamos a ver con un poco más de profundidad los contenedores asociativos que son los que usaremos al implementar las relaciones entre clases de objetos en C++.

1.2.1. Conjunto de objetos: Set

El contendor asociativo `set` es un conjunto de elementos no duplicados de un mismo tipo. Pueden estar ordenados (`set`) ó desordenados (`unordered_set`).

Librería → `<set>`, donde se incluye todos los métodos de los que puede hacer uso mediante iteradores.

Sintaxis conjunto valores no repetidos:

```
typedef std::set<T>Alias;
```

Se hace uso de Alias cuando se define en la parte pública de la clase, si fuera en la privada, ese sería el nombre del contenedor.

Sintaxis conjunto valores repetidos:

```
typedef std::multiset<T>Alias;
```

El tipo `T` definido en el set puede ser un **tipo de dato (objeto)** o un **puntero**. Si el tipo es un objeto tendremos que realizar una sobrecarga de un operador de ordenación, ya sea el operador `<` o el operador `>` para ordenadar los elementos del conjunto, esto es lo que sucede a la hora de implementar la **relación de composición** entre dos clases.

Si es de **tipo puntero**, no hace falta realizar esta sobrecarga para ordenar los elementos.

Operaciones del contenedor asociativo set:

Vamos a ver las operaciones más comunes que usaremos a la hora de implementar un conjunto de elemento en las relaciones entre clases.

```
#include <set>
set<T> conjunto;
/*-----------INSERCION------------*/
conjunto.insert(const T& e); // inserta un elemento en el conjunto.
/*----------ELIMINACION----------*/
conjunto.erase(iterator p); // elimina el elemento que se encuentre
    en la posicion.
conjunto.clear(); // elimina todos los elementos del conjunto
/*----------ACCESO---------- */
conjunto.begin(); //devuelve iterador al primer elemento del
   conjunto.
conjunto.end(); //devuelve iterador siguiente del ultimo elemento
   del conjunto.
conjunto.find(const T& e); //Devuelve el iterador que apunta ese
   elemento.
/*----------CONSULTA---------- */
conjunto.empty(); //Devuelve V o F si esta vacio o no.
conjunto.size(); //Devuelve el tam del conjunto
```

1.2.2. Diccionario clave - valor: Map

El contenedor asociativo `map` es un diccionario de pareja clave única y valores o conjunto de valores. Al igual que el contendor asociativo *set*, `map` también es un contenedor donde las claves se insertan de manera ordenada, si no queremos un diccionario ordenado, podemos hacer uso de `unordered_map`.

Librería → `<map>`, donde se incluye todos los métodos de los que puede hacer uso mediante iteradores, al igual que `set`.

Sintaxis clave - valor:

```
typedef std::map<T1,T2>Alias;
```

Aqui vemos que la declaración se realiza con una clave y un valor que puede tomar cualquier valor (T1 y T2, si estuvieramos en una plantilla).

Sintaxis clave - conjunto valores:

```
typedef std::multimap<T1,T2>Alias;
```

Este `multimap` lo podemos implementar de una manera más intuitiva:

```
typedef std::map<T1,conjunto>Alias;
```

A la hora de declararlo mediante conjunto de elementos dento de un `map`, dicho conjunto puede ser un `set`, `unordered_set`, `map`, `unordered_map` o `pair`.

Se hace uso de Alias cuando se define en la parte pública de la clase, si fuera en la privada, ese sería el nombre del contenedor.

Mediante los **iteradores** podemos acceder tanto a la clave (`first`) como al valor (`second`).

Operaciones del contenedor asociativo map:

Vamos a ver las operaciones más comunes que usaremos a la hora de implementar un diccionario en las relaciones entre clases.

```
#include <map>
map<T,U>diccionario;
/*-----------INSERCION------------*/
//inserta para una clave un valor
diccionario.insert(std::make_pair(const T& clave, const U& valor));
diccionario[const T& clave] = &valor; //permite modificar el valor
   si ya hay un valor previo insertado en esa clave.
/*-----------ELIMINACION------------*/
diccionario.erase(iterator p); //elimina la clave de la posicion p.
diccionario.erase(); //elimina todo el diccionario
/*-----------ACCESO------------*/
diccionario.begin(); //Devuelve iterador a la primera clave del
   diccionario.
diccionario.fin(); //Devuelve iterador a la siguiente posicion de
   la ultima clave del diccionario.
diccionario[const T& clave];//accede al valor de la clave dada
/*----------CONSULTA---------- */
diccionario.empty(); //Devuelve V o F si esta vacio o no.
diccionario.size(); //Devuelve el tam del conjunto
```

1.2.3. Tupla de dos elementos: Pair

Esta es una clase genérica la cual permite crear pareja de valores únicos de cualquier tipo T.

Librería → `<utility>`, una cosa a tener en cuenta es que las parejas de elementos no tienen los métodos `insert()`, `erase()`, por lo que a la hora de insertar lo haremos mediante asignación.

Sintaxis tupla de valores:

```
typedef std::pair<T1,T2>Alias;
```

Siendo T1 y T2 los tipos de los dos elementos que almacena.

Este tipo de tuplas son muy útiles cuando encontramos multiplicidades 1 - muchos en las relaciones entre clases y la podemos declarar dentro de los diccionarios `map`.

1.3. Algoritmos de búsqueda y ordenación

Para poder buscar elementos en un contenedor podemos hacer uso de `find`, `binary_search`. Podemos ordenar los elementos de los contenedores mediante los algoritmos `sort`.

1.4. Algoritmos de modificación

Podemos realizar varias modificaciones en los contenedores mediante varios algoritmos como `std::copy` (donde hacemos una copia de todos los elementos de un contenedor en otro), `std::fill` (rellenamos el contenedor con elementos), `std::move` (donde movemos el contenido de un contenedor a otro, se realiza mediante la semántica de movimiento de C++.)

1.5. Funciones Lambda y Objetos a Función

Encontramos varios *algoritmos de la STL* que para su ejecución necesitan que se satisfaga una condición o predicado. Estos predicados pueden ser tanto **funciones lambda** ó **objetos a función**.

1.5.1. Funciones Lambda en C++

Permiten definir funciones anónimas de manera concisa y local. Las funciones *Lambda*proporcionan una forma de crear funciones en el lugar donde se necesitan, sin tener que definir una función por separado.

Consiste en tres partes → lista de captura `[]` , parámetro opcional de captura `()` y un cuerpo {}.

Cuando se evalúa una función *Lambda*, se produce un objeto llamado *closure*. Este objeto '*closure*' no tiene ni nombre ni tipo y es temporal.

```
auto nombre_funcion = [captura](parametros){
 //cuerpo de la funcion
 // Puede acceder a las variables capturadas y usar parametros
 return expresion;
}
```

Tipo de retorno → Especifica el tipo de dato que la función *Lambda*devolverá. Puede ser omitido si esta no devuelve un valor explícitamente.

Captura → Permite a la función lambda acceder a las variables fuera de su alcance local. Puede ser por valor `[x]` , por referencia `[&x]` , por valor pero modificable `[x] mutable` , por referencia (solo lectura) `[&x]mutable` o capturar todas las variables automáticamente `[=]` ó `[&]`.

Parámetros → Los parámetros de la función *Lambda* son similares a los usados en las funciones regulares. Si no tiene parámetros '`()`' estos paréntesis serán omitidos a no ser que se necesites declarar la función *Lambda* como `mutable`.

Cuerpo de la función→ El '*body*' contiene la lógica de la función *Lambda* (como una función regular).

Expresión → Valor de la función *Lambda* puede devolver (opcional si la función no devuelve nada o si el tipo de retorno es `void`).

Ejemplo del uso de función Lambda

```
int main(){
//Funcion Lambda que suma dos numeros
  auto suma = [](int a, int b){return a+b;}
//Uso de la funcion Lambda
  int resultado = suma(5,7);
  std::cout<<"Resultado: "<<resultado<<std::endl;
  return 0;
}
```

Vemos que en este ejemplo, la función Lambda `suma` toma dos parámetros 'a' y 'b' de tipo enteros ambos y devuelve el resultado de la suma de ambos. La función se almacena en una variable `auto` y luego se utiliza como cualquier otra función.

Ejemplo del uso de función Lambda mutable

La *keyword* `mutable` permite que la función Lambda pueda ser modificada, ya que estas por defecto son no modificables `const`.

```
int main(){
  int a =0;
  //Error, operator () es const
  auto bad_counter = [a](){return a++;}
  //Correcto, operator () es mutable
  auto good_counter = [a]()mutable{return a++;}
  //Llamamos al metodo
  good_counter(); //a=1
  good_counter(); //a=2;
  return 0;
}
```

1.5.2. Objeto Función en C++

Los **objetos función** o **funtores**: Son objetos que se comportan como funciones. Es una instancia de una clase que sobrecarga el operador `operator()`.

Definición mediante clases: Para crear un objeto función, necesitas definir una clase que tenga el operador de llamada sobrecargado.

```
struct Suma{
  int operator() (int a , int b) const {return a+b;}
};

int main(){
  Suma objetoFuncion;
  //creamos una variable con el resultado de llamar al objeto a
     funcion
  int resultado = objetofuncion(3,4);
  return 0;
}
```

1.6. Algoritmos de la STL

Mediante lo visto anteriormente podemos hacer uso de algoritmos para poder manipular el contenido de los contenedores de la *STL*. Hay algunos algoritmos que haran uso de los **iteradores**, otros el uso de **predicados** o ambos.

1.6.1. std::count_if

Lo encontramos en la librería `<algorithm>`. Devuelve el número de veces que se repite un elemento en el rango [inicio,fin) de un contenedor.

```
std::count_if(iterador_ini,iterador_fin,predicado);
```

1.6.2. std::remove_if

Lo encontramos en la librería `<algorithm>`.

Elimina el elemento que cumple la condición dada por el predicado en el rango [inicio,fin) del contendor.

```
std::remove_if(iterador_ini,iterador_fin, predicado);
```

1.6.3. std::lower_bound

Lo encontramos en la librería `<algorithm>`. Devuelve un iterador apuntando a la primera posición donde el elemento que se le pasa es '$<$' que el siguiente elemento. Es muy útil para la inserción o búsqueda de elementos de un contenedor ordenado.

```
std::lower_bound(ini_rango,fin_rango,const T& elto);
```

1.6.4. std::insert

Este método está contenido en todas las librerías de los contenedores de la *STL* como **vector**, **map**, **set** y los demás tipos contenedores excepto `std::pair`, como este no tiene el método `insert`, la inserción se hará por asignación. Inserta el elemento especificado dado un iterador, es decir, colocará dicho elemento en la posición del contenedor que le especifiquemos.

```
std::insert(const_iterator pos, const T& elto);
```

1.6.5. std::erase

Este método al igual que `std::insert` se define en cada contenedor de la *STL*. Elimina un elemento específico que se encuentre contenido en un contenedor dando su posición (iterador).

```
std::erase(const_iterator pos);
```

1.6.6. std::find

Se encuentra en la librería `<algorithm>`. Devuelve un iterador al primer elemento del rango [inicio,fin) que satisface un criterio específico.

Muy útil para buscar elementos contenidos en un tipo contenedor.

```
auto i = nombre_contenedor.find(T& elto);
```

2. Introducción a las relaciones

Podemos diseñar relaciones entre los objetos de las diferentes clases de objetos. Estas relaciones pueden ser muy diferentes entre sí, habiendo algunas más estrictas que otras. Por defecto

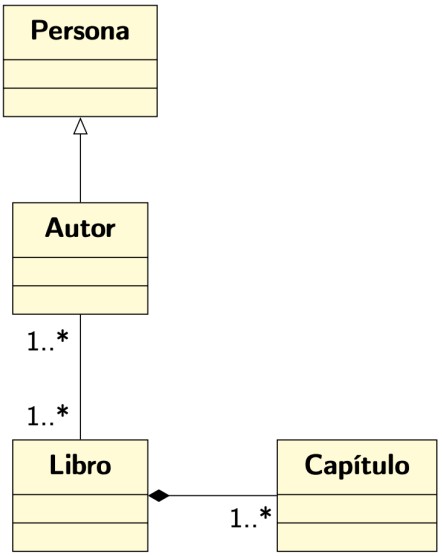


Figura 2.1: Diagrama de Clases

Al conjunto de relaciones entre las diferentes clases de datos, lo denominamos →*diagramas de clases*.

Aquí encontramos varios tipos de relaciones (asociaciones, generalizaciones, composiciones, etc.).

Entre las diferentes clases encontramos enlaces de objetos, los cuales contienen una multiplicidad y dirección.

Los objetos se **enlazan** y las clases se **relacionan**.

la relación es **bidireccional** por tanto, tenemos que incluir la relación en ambas clases, si no fuera así (es decir unidireccional) solamente se define la relación y los métodos necesarios en la clase opuesta a la "flecha".

3. Asociaciones entre clases

Se representa mediante una línea continua entre las clases asociadas. En este caso podemos saber que dado un libro podemos saber sus autores o viceversa. Las asociaciones entre clases contienen 3 factores (cardinalidad, navegabilidad y multiplicidad).

- La **cardinalidad** es el número de clases asociadas, por defecto, son binarias (cardinalidad = 2).
- La **multiplicidad** indica cuántos objetos de las clases se enlazan, en este caso como mínimo se enlaza 1 y como máximo muchos (1..*). Si no se especifica es 1.
- La **navegabilidad** indica el sentido de la relación, por defecto, son bidireccionales pero pueden existir asociaciones de $A \rightarrow B$ o $A \leftarrow B$ (unidireccionales, donde el extremo de la flecha es el sentido).

Estas asociaciones pueden tener algunas dependencias (agregaciones y composiciones).

3.1. Implementaciones

Dependiendo de la multiplicidad que haya en cada extremo de las clases de objetos implemetaremos de una manera u otra las relaciones entre las clases.

Donde las relaciones que tienen una multiplicidad muchos - muchos se harán uso de tipos contendores asociativos o si la multiplicidad es 1 - 1 solamente se almacenará el objeto con el que se enlazan.

3.1.1. Plantilla Implementación 1-N

```
class B; //declaracion adelantada
class A{ //clase con multiplicidad 1
public:
  //...
  typedef std::set<B*>Bs;
  void setB(B& b){bs_.insert(&b);}; //guardamos los elementos en
     el conjunto
  const Bs& getA() const{return bs_;}; //objeto B enlazado con A
private:
  Bs bs_; //enlace con objeto B
};
class B{ //clase con multiplicidad N
public:
  //...
  void setA(A& a){a_ = a;}
  const A& getB()const{return a_;}
private:
  A a_;
};
```

3.1.2. Implementación 1-1

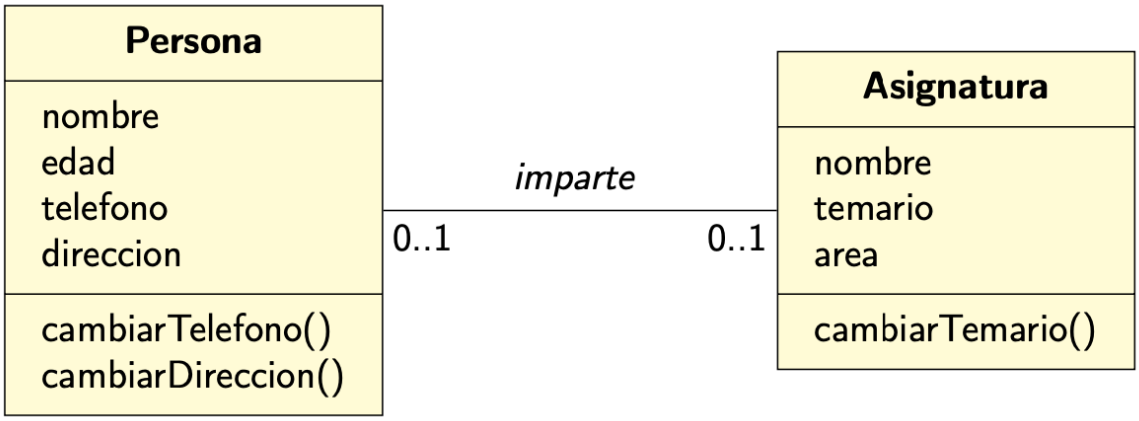


Figura 3.1: Asociación 1-1

```
class Asignatura; //declaracion adelantada
class Persona{
  public:
  //...
  void setA(Asignatura& a){asig_ = &a;}; //enlazamos
  //objeto Asignatura enlazado con Persona
  const Asignatura& getA() const noexcept {return *asig_;}
  private:
  Asignatura* asig_; //enlace con objeto Asignatura
};
class Asignatura{
  public:
  //...
  void setP(Persona& p)noexcept{persona_ = &p;} //enlazamos
  const Persona& getP()const noexcept{return *persona_;} //objeto
     Persona enlazado con Asignatura
  private:
  Persona* persona_; //enlace con objeto Persona
};
```

3.1.3. Implementación N-M

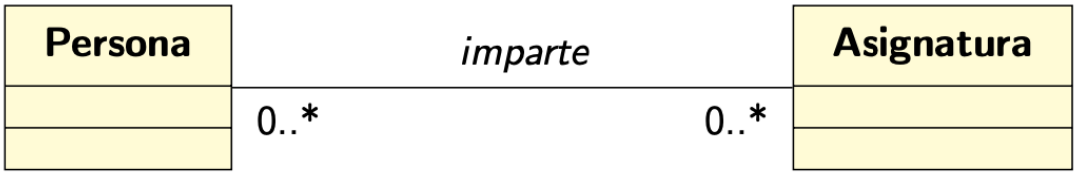


Figura 3.2: Asociación muchos - muchos

```
class Asignatura;
class Persona{
  public:
    //...
    typedef set<Asignatura*> As;
    void setB(Asignatura& a)noexcept{as_.insert(&a);}
    const As& getA()const noexcept{return as_;}
  private:
    As as_;
};

class Asignatura{
  public:
    //...
    typedef set<Persona*> Ps;
    void setP(Persona& p)noexcept{ps_.insert(&p);}
    const Ps& getB()const noexcept{return ps_;}
  private:
    Ps ps_;
};
```

4. Agregación y Composición

La diferencia entre agregación ('diamante vacío', menos restrictiva) y composición ('diamante relleno', más restrictiva). Véase en Libro y Capítulo (Introduccion a las relaciones), donde un Libro no puede existir si no existen Capítulos (composición).

La composición es un tipo de agregación donde existe dos restricciones, de **multiplicidad** (un componente solo puede formar parte de un compuesto) y **existencia** (la vida del componente depende del compuesto).
Mediante las composiciones también hacemos uso de la reutilización del clases, donde hacemos uso de la clase Capítulo para definir la clase Libro.

4.1. Relación de agregación

> Es la relación de asociación entre una clase y un agregado (conjunto).

Normalmente, son unidireccionales, la clase componente no conoce nada de la clase compuesta por ella misma.

Se implementan como relaciones de **asociación** pero son unidireccionales donde solamente almacena el contenido la clase que es compuesta (la tiene el diamante).

4.2. Relación de Composición

> Es la relación de asociación entre un componente y un compuesto, pero es más restrictiva que la relación de agregación.

El atributo que añadimos a la clase compuesta, ya no es un puntero, si no el objeto como tal.

Si la multiplicidad es $1 \rightarrow$ solamente un atributo, si no, tendremos un contenedor del tipo de la clase componente. Podemos hacer uso de estas relaciones para la reutilización de software.

4.2.1. Implementación de composición 1 - 1..N

```
class B; //declaracion adelantada
class A{ //Compuesto
  public:
    //A se crea a partir de un objeto de B
    A(B& b){ setB(b); }
    typedef std::set<B> BS;
    void setB(const B& b)noexcept{bs_.insert(b);}

  private:
    BS bs_;
};
class B{//Componente
//No recibe nada del compuesto
  public:
    friend bool operator < (const B& a, const B& b) noexcept;
};
```

En este tipo de relaciones al tener un `set` de objetos de la clase *componente* necesitamos definir una manera de ordenar los elementos que componen, por eso sobrecargamos el operador menor que '<'.

4.2.2. Implementación de composición 1 - 1

```
class A{ //Compuesto
 public:
   A(B& b):b_(b){} //A se crea a partir de un objeto de B
 private:
   B b_;
};
class B{//Componente
   //No recibe nada del compuesto
};
```

4.2.3. Implementación de composición 1 - 1 (herencia privada)

```
class A : private B{ //Compuesto
 public:
   A(B& b): B(b){} //A se crea a partir de un objeto de B
};
class B{//Componente
   //No recibe nada del compuesto
};
```

5. Asociación Calificada

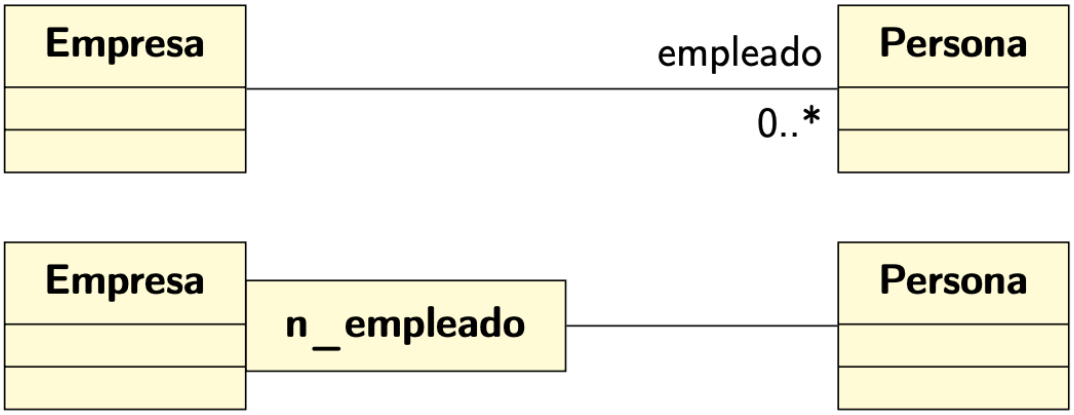


Figura 5.1: Relaciones calificadas

Mediante esto, reducimos la multiplicidad en el lado muchos, haciendo las búsquedas mucho más eficientes. Solamente podemos hacer uso de un atributo que sea *clave primaria* de la clase. Pasamos de 1 - muchos a 1 - 1.

Lo implementamos mediante un diccionario en la clase empresa. Si la multiplicidad no se reduce, significa que dicho calificador no es **clave primaria** y por tanto, se pueden obtener diferentes valores para dicha clave.

5.0.1. Implementación de la relacion

```
class Empresa{
 public:
//mediante un calificador,
//encontramos un objeto de la clase Persona
   typedef std::map<T,Persona*>Bcalificada;
//metodos de la clase
   void asocia(Persona&);
   const Bcalificada& getA()const;
 private:
   Bcalificada bc_;
};
class Persona{
 public:
//metodos propios de la clase
   typedef std::set<Empresa*>As;
   void asocia(Empresa&);
   const As& getB()const;
 private:
   T calificador; //calificador de tipo T(cualquier tipo de dato)
   As as_;
};
```

6. Asociación con atributo de Enlace

Es una relación de asociación donde encontramos un atributo que se enlaza con la relación de las clases A - B. Ese atributo no pertenece ni a la clase A ni a la clase B, si no a la relación.

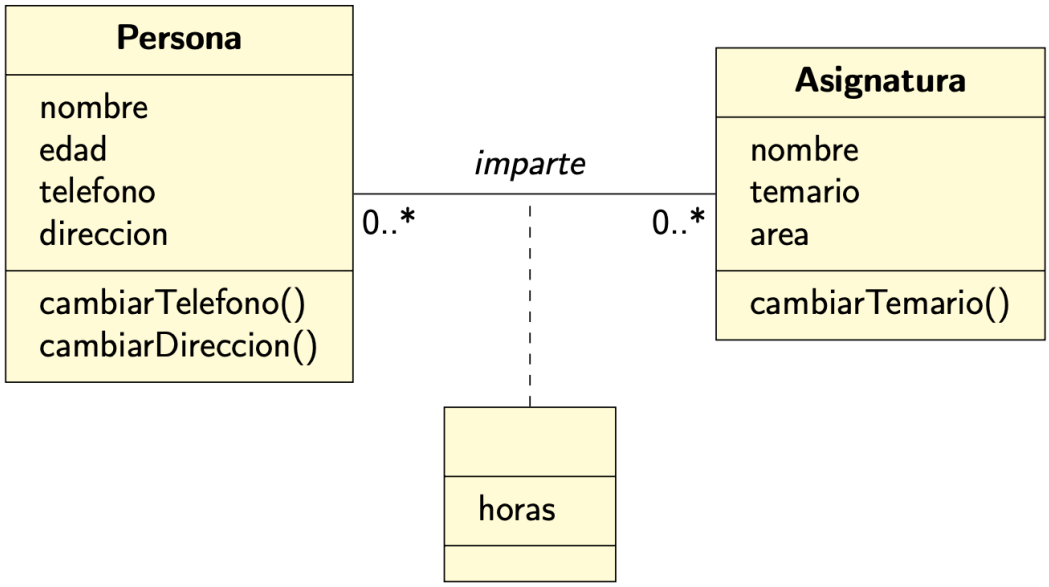


Figura 6.1: Asociación con atributo de enlace

Vemos que una persona imparte asignaturas un número de horas y las asignaturas son impartidas por personas un número de horas.

Dependiendo de la multiplicidad podemos hacer que dicho atributo pertenezca a una de las dos clases (la que tiene mayor multiplicidad).

Si Asignatura tuviera multiplicidad 1, el atributos *horas* iría en la clase Persona → hacemos uso de un conjunto '`set`'.

En este caso, ambos tienen multiplicidad muchos - muchos, por tanto, ese atributo no puede almacenarse en ninguna de las dos clases → tenemos que hacer uso de un diccionario '`map`'.

6.0.1. Implementación de la relación:

```
class Asignatura;
class Persona{
  public:
    typedef std::map<Asignatura*,int> Bs;
    void setA(Asignatura&, int) noexcept;
    const Bs& getB()const noexcept;
  private:
    Bs bs_;
};

class Asignatura{
  public:
    typedef std::map<Persona*,int> As;
    void setB(Persona&, int)noexcept;
    const As& getA()const noexcept;
  private:
    As as_;
};
/*----------Implementacion de los metodos----------*/
void Persona::setA(Asignatura& a, int atributo) noexcept{
  //Sin permitir que se modifique la clave si existe en el
     diccionario.
  bs_.insert(std::make_pair(&a,atributo));

  //Permite modificar la clave si ya existe en el diccionario.
  //bs_[&a]=atributo;
}

const Persona::Bs& Persona::getB()const noexcept{
  return bs_;
}

void Asignatura::setB(Persona& p, int atributo)noexcept{
  //No permite que se modifique la clave si existe en el
     diccionario.
  as_.insert(std::make_pair(&p,atributo));

  //Permite modificar la clave si ya existe en el diccionario.
  //as_[&p]=atributo;
}

const Asignatura::As& Asignatura::getA()const noexcept {
  return as_;
}
```

7. Asociación con Clase Asociación

Si en el caso de una relación de asociación con atributo de enlace, este contiene varios atributos estamos ante una clase de asociación.

Esta clase de asociación podemos llevarla a cabo si tenemos una asociación muchos - muchos entre dos clases, tenemos varios atributos de enlace o tenemos varias clases asociadas.

7.0.1. Implementación clase asociación con 3 clases

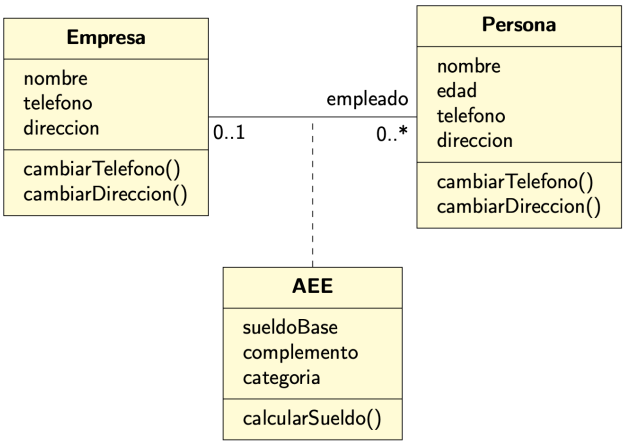


Figura 7.1: Asociación con Clase de Asociación

```
class Empresa{}; class Persona{}; class Salario{};
class AEE{//clase de asociacion resultante
  public:
    typedef std::map<Empresa*,std::map<Persona*,Salario*>>AD;
    typedef std::map<Persona*,std::pair<Empresa*,Salario*>>AI;
    void setA(Empresa&,Persona&,Salario&) noexcept;
    void setB(Persona&,Empresa&,Salario&) noexcept;
    //devolvemos una copia, ya que pueden no haber instancias
       relacionadas.
  const std::map<Persona*,Salario*> getB(Empresa& )const noexcept;
  const std::pair<Empresa*,Salario*>* getA(Persona &)const noexcept
     ;
  private:
    AD directa_;
    AI inversa_;
};
```

Si la multiplicidad en una de las dos clases es 0..1, podemos hacer uso de '`pair`' en vez de `map`. En este caso, como empresa es 0..1, lo implementamos de mediante pair.

Si la clase de asociación tienen multiplicidad 1..N - 1..M en los observadores devolveremos una referencia constante ya que siempre habrá al menos un objeto instanciado.

7.0.2. Implementación clase de asociación con 2 clases

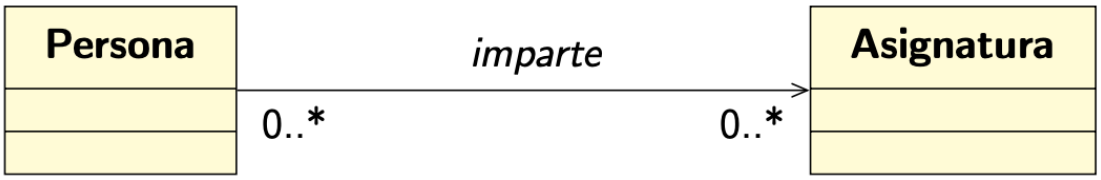


Figura 7.2: Asociación muchos - muchos con Clase de Asociación

También es una alternativa para implementar la asociación entre dos clases, siempre y cuando la multiplicidad sea muchos - muchos o no tengamos el código fuente de ambas clases y no podamos incluir los métodos propios de la asociación.

Podemos crearnos una clase nueva, que representa la relación entre estas dos clases.

```
class Persona{};class Asignatura{};
class PersonaAsignatura{
  public:
    typedef std::multimap<Persona*,Asignatura*>AD;
    //otra manera -> typedef std::map<Persona*,std::set<Asignatura
       *>> AD;
    typedef std::multimap<Asignatura*,Persona*>AI;
    //typedef std::map<Asignatura*,std::set<Persona*>>AI;
    void setPersona(Persona&,Asignatura&);
    void setAsignatura(Asignatura&,Persona&);
    void getPersona(Asignatura&)const;
    void getAsignatura(Persona&)const;
    //std:set<Persona*>& getPersona(Asignatura&)const;
    //std::set<Asignatura*>& getAsignatura(Persona&)const;
 private:
    //nos ayuda a insertar nuevos pares.
    bool esta(Persona&,Asignatura&)const;
    AD directa;
    AI inversa;
};
```

Para prevenir la inserción múltiple de valores a una clave tenemos 2 maneras:

1. Si hacemos uso de `multimap` tenemos que declarar el método `esta()` para comprobar que dicho valor no está insertado.
2. Podemos hacer uso de un `set` dentro de un `map`, haciendo que por cada clave puede haber un único conjunto de valores → `std::map<A*,std::set<B*>>Nombre`;

Para ahorranos tener que escribir la declaración entera del `set`, podemos crear un alias `typedef std::set<...>Alias;`

```
bool PersonaAsignatura::esta(Persona& p, Asignatura& a) const{
//std::pair<PersonaImparteAsignatura::ID,
//         PersonaImparteAsignatura::ID>
//         rango = directa.equal_range(&p);
auto rango = directa.equal_range(&p);//devuelve un rango <first,
   second>
for (PersonaImparteAsignatura::ID i = rango.first; i != rango.
   second; ++i)
    if (i->second == &a) return true;
  return false;
}
```

En el método `esta()` tenemos que buscar todas las ocurrencias de dicha clave, por tanto, `find` no nos sirve ya que nos devuelve la primera y necesitamos todas. Por eso, usamos `equal_range(&clave)` para poder obtener un rango (la primero y última ocurrencia que tiene esa clave).
Para poder obtener el conjunto de personas o de asignaturas mediante un set dentro de un map:

```
std::set<Asignatura*> PersonaAsignatura::getAsignatura(Persona& p)
   const{
  auto i = directa.find(&p);
  if(i!=directa.end())return i->second;
  else return std::set<Asignatura*>();//devolvemos conjunto vacio.
}

std:set<Persona*> PersonaAsignatura::getPersona(Asignatura& a)const
   {
  auto i = inversa.find(&a);
  if(i!=inversa.end()) return i->second;
  else return std::set<Persona*>(); //devolvemos conjunto vacio.
}
```

La devolución del set que es un conjunto vacío lo va a realizar mediante movimiento, el conjunto devuelto normal será una copia.

Ejemplo:

```
//Personas es un Alias de: typedef std::set<Persona*>Personas;
/*-------------------------------------------------*/
PersonaAsignatura::Personas PersonaAsignatura::getPersona(
   Asignatura& a)const{
  auto i = inversa.find(&a);
  if(i!=inversa.end()) return i->second;
  else{
    PersonaAsignatura::Personas vacio;
    return vacio;
  }
```

7.0.3. Clase genérica de asociación con 2 clases

```
template <typename X, typename Y>
class AsociacionBidireccional{
public:
  typedef std::set<Y*>Ys;
  typedef std::set<X*>Xs;
  typedef std::map<X*, Ys> Directa;
  typedef std::map<Y*, Xs> Inversa;
  void asocia(X& x, Y& y);
  void asocia(Y& y, X& x);
  Ys asociados(X& x) const;
  Xs asociados(Y& y) const;
private:
    Directa directa_;
    inversa inversa_;
//-----Implementacion de los metodos-----
template <typename X, typename Y>
void AsociacionBidireccional<X, Y>::asocia(X& x, Y& y){
    directa_[&x].insert(&y);
    inversa:[&y].insert(&x);
}
template <typename X, typename Y>
inline void AsociacionBidireccional<X, Y>::asocia(Y& y, X& x)
{ asocia(x, y); }

//Devuelve el conjunto de enlaces asociados a un objeto .
template <typename X, typename Y> AsociacionBidireccional<X,Y>::Ys
 AsociacionBidireccional<X, Y>::asociados(X& x) const
{
  auto i = directa.find(&x);
  if (i != directa.end()) return i->second;
  else{
    //return std::set<Y*>();
    AsociacionBidireccional<X, Y>::Ys vacio;
    return vacio;
  }
}
template <typename X, typename Y> AsociacionBidireccional<X,Y>::Xs
  AsociacionBidireccional<X, Y>::asociados(Y& y) const
{
  auto i = inversa.find(&y);
  if (i != inversa.end()) return i->second;
  else{
    //return std::set<X*>();
    AsociacionBidireccional<X, Y>::Xs vacio;
    return vacio;
  }
}
```

7.0.4. Clase genérica de asociación con atributo de enlace

```
template <typename X, typename Y, typename Z>
// X e Y: clases asociadas
// Z: clase de los atributos de enlace
class AsociacionBidireccional {
public:
  void asocia(X& x, Y& y, Z& z);
  void asocia(Y& y, X& x, Z& z);
  map<Y*, Z*> asociados(X& x) const;
  map<X*, Z*> asociados(Y& y) const;
private:
  map<X*, map<Y*, Z*> > directa;
  map<Y*, map<X*, Z*> > inversa;
};
//-----Implementacion de los metodos-----

// Asocia bidireccionalmente dos objetos .
template <typename X, typename Y, typename Z>
void AsociacionBidireccional<X, Y, Z>::asocia(X& x, Y& y, Z& z){
  //directa[&x][&y]=(&z); //permite modificar z
  directa[&x].insert(make_pair(&y, &z));
  //inversa[&y][&x]=(&z); //permite modificar z
  inversa[&y].insert(make_pair(&x, &z));
}
template <typename X, typename Y, typename Z> inline
void AsociacionBidireccional<X, Y, Z>::asocia(Y& y, X& x, Z& z)
 { asocia(x, y, z); }

// Devuelve el conjunto de enlaces asociados a un objeto .
template <typename X, typename Y, typename Z>
map<Y*,Z*> AsociacionBidireccional<X, Y, Z>::asociados(X& x) const
{
  map<X*, map<Y*, Z*>>::const_iterator i = directa.find(&x);
  if (i != directa.end()) return i->second; //por valor
  else return map<Y*, Z*>(); //diccionario vacio por movimiento
}
template <typename X, typename Y, typename Z>
map<X*,Z*> AsociacionBidireccional<X, Y, Z>::asociados(Y& y) const
{
  map<Y*, map<X*, Z*>>::const_iterator i = inversa.find(&y);
  if (i != inversa.end()) return i->second;
  else return map<X*, Z*>();//diccionario vacio por movimiento
}
```

8. Generalización y Especialización

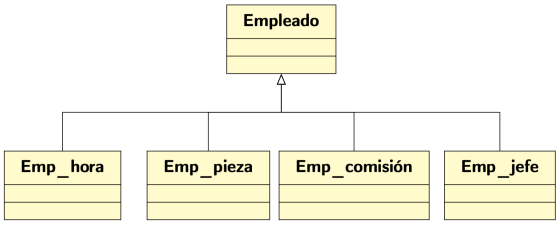


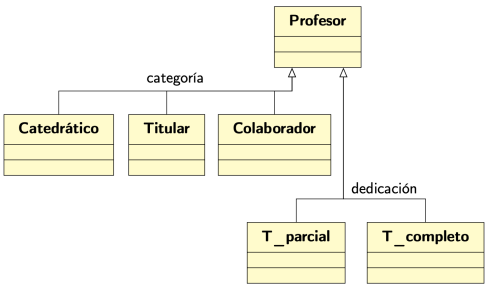
Figura 8.1: Diagrama de Clases

Relación 'ES-UN' que existe entre una *superclase* y una *subclase*.
Una subclase contiene las mismas características que la superclase y atributos extras de la misma.
Es lo que en C++ denominamos como herencia, donde la subclase *Emp_hora*hereda todos los atributos y métodos de la superclase *Empleado* y además estas subclases tendrán métodos propios que no contendrá la superclase.
Las generalizaciones pueden ser *simples* o *múltiples.*

Las subclases se **generalizan** en una súperclase, al revés la súperclase se **especializa** en subclases.

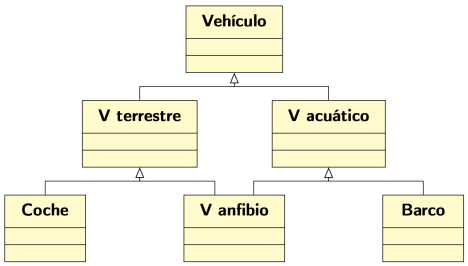
8.1. En cuanto al diseño

8.1.1. Generalización mediante criterios



Podemos generalizar mediante condiciones 'discriminador'. Donde podemos ver que un *profesor* puede tomar diferentes categorías y diferente dedicación.

Ejemplo de Generalización múltiple:
Vemos que un vehículo puede ser terrestre o acuático y a su vez los terrestres pueden ser un coche o un anfibio, y los acuáticos pueden ser anfibio o barcos.



8.2. En cuando a la implementación

La programación Orientada a Objetos nos permiten hacer uso de **herencia** para poder implementar este tipo de relaciones.
La herencia es una herramienta muy potente, ya que nos permite la reutilización de código, ya que podemos definir nuevas clases a partir de otras ya definidas.
Pero esto puede ser un peligro, ya que podemos confundir una asociación con una **generalización-especialización**.

Podemos hacernos varias preguntas para saber cual tipo de relación escoger:

- ¿Es la clase **x** un tipo especial de la clase **y**?
- ¿Es posible usar un objeto de la clase **x** en todas aquellas situaciones en que se emplee un objeto de la clase **y**?
- ¿Y si un objeto de la clase **x** se va a ejecutar simultáneamente en dos o más relaciones con la clase **y**, cada una mostrando una vista diferente del mismo?

```
class clase_derivada[final]:
[accesibilidad] clase_base{
//Declaracion de los miembros de
   la clase
};
```

Una clase derivada **NO** hereda:
- Constructores
- Destructor
- Operadores de asignación

Si ponemos final, estamos indicando que esa clase no se va a poder derivar más.

La accesibilidad puede ser `public` , `private` ó `protected`.

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

- `Public` → Accesible desde el interior y exterior.
- `Protected` → Accesible desde el interior, para funciones amigas de C y desde las clases derivadas de C y sus funciones amigas. Sirven para que los atributos sea accesibles a las clases derivadas pero ocultos al exterior.
- `Private`→ Accesible desde el interior y para las funciones amigas de C.

Además `protected` hace que no se cumpla el principio de ocultación de información y las clases estén acopladas, es decir, que debemos comprobar que las clases funcionan para los mismos tipos de datos.

Accesibilidad de la herencia	Un miembro... de la clase base	pasa a ser... en la derivada
`public` (por omisión en `struct`)	público protegido privado	público protegido inaccesible
`protected`	público protegido privado	protegido protegido inaccesible
`private` (por omisión en `class`)	público protegido privado	privado privado inaccesible

Figura 8.3: Modificación de accesibilidad de miembros heredados

8.3. Herencia pública

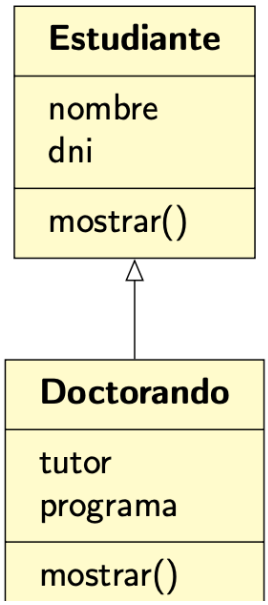
Nos sirve para implementar las relaciones de especialización/generalización, es decir, las relaciones ES-UN entre clase base y subclases. Hereda su comportamiento externo.

Esta relación al ser pública, es visible desde el exterior, por tanto, *un objeto de una subclase es también un objeto de la clase base. Podemos realizar conversiones hacia arriba, un objeto de la clase derivada puede convertirse en uno de la clase base, debido a que la clase derivada contiene la misma información que la clase base.* Pero **no** se *puede convertir un objeto de la clase base en uno de la derivada debido a que a esta le falta información específica de las clases derivadas.*

8.3.1. Ejemplo de Herencia Pública:

Vemos que un *Doctorando* es una especialización de la *clase Estudiante.*
Donde la clase *Doctorando* tendría 4 atributos (2 propios y 2 heredados de la clase base) y además un método llamado `mostrar()`.

Si no ponemos los atributos con los que se crea un objeto de la clase base y este no tiene constructor predeterminado, nos dará un error.




```
class Estudiante{
  public:
    Estudiante(string nombre, int dni);
    void mostrar() const;
  private:
    string nombre; int dni;
    // ...
};

class Doctorando final : public Estudiante{
//doctorando no se puede derivar mas
  public:
    Doctorando(string nombre,int dni,
          string tutor, int programa);
    void mostrar() const;
  private:
    string tutor;
    int programa;
    // ...
};
```

```
Doctorando::Doctorando(string nombre, int dni, string tutor, int
   programa):Estudiante(nombre, dni), tutor(tutor), programa(
   programa) {}

void Doctorando::mostrar() const{
  // Muestra los datos que posee como estudiante
  Estudiante::mostrar();
  // Y los especificos de su condicion de doctorando
  cout << "Programa de doctorado: " << programa << "\n"
       << "Tutor en el programa: " << tutor << endl;
}
```

Llamamos al método mostrar que está definido en estudiante, para poder mostrar información sobre el estudiante sea cual sea su especialización.

Si se quiere mostrar la información del *Estudiante* deberíamos de definir observadores debido a que los atributos son **privados**.

Esto sucede debido a que estamos dentro del ámbito de *Doctorando* y no de *Estudiante*.

Si quitamos el operador de resolución de ámbito, tendríamos una función recursiva.

```
Estudiante* pe= &e; //puntero a estudiante
pe->mostrar(); //metodo mostrar de estudiante
pe=&d; //conversion hacia arriba de doctorando a estudiante
pe->mostrar(); //metodo mostrar de estudiante
/*-----------------------------------------------------*/
Doctorando* pd = &d; //puntero a doctorando
pd->mostrar(); //metodo mostrar de doctorando
pd->Estudiante::mostrar(); //metodo mostrar de estudiante (
   resolucion ambito)
pd->Doctorando::mostrar(); //metodo mostrar de doctorando.

/*-----Conversiones entre punteros-----*/
pe = &d; //conversion implicita entre punteros hacia arriba
pd = &e; //Error -> no se puede hacer una conversion implicita
   hacia abajo
pd= static_cast<Doctorando>(e);//Conversion en T.Compilacion
   corrige lo anterior

/*-----Conversiones entre objetos-----*/
e = d; //Objeto d Doctorando se convierte a un tipo Estudiante
d = e; //Error -> No se puede convertir un objeto e Estudiante (
   base) a d Doctorando (derivado)
//Tampoco se puede hacer explicita la conversion hacia abajo entre
   objetos:
        d = Doctorando(e); //Error
        d = static_cast<Doctorando>(e);//Error
        d = reinterpret_cast<Doctorando>(e);//Error
```

Figura 8.4: Ejemplo de herencia pública

No se pueden realizar conversiones implícitas hacia abajo entre punteros, estás deben de ser explicitas.
No se pueden realizar conversiones implícitas ni explícitas hacia abajo entre objetos de las clases base y derivada.

8.4. Herencia privada

Sirven para implementar relaciones "Se implementan con". Se hereda la interfaz pública de la base pero en la clase derivada se ocultan debido a que estos pasan a ser **privados**. Por tanto, la relación entre ambas clases queda oculta para el usuario.

Se puede usar para implementar una ***relación de composición 1a1*** entre componente y compuesto.

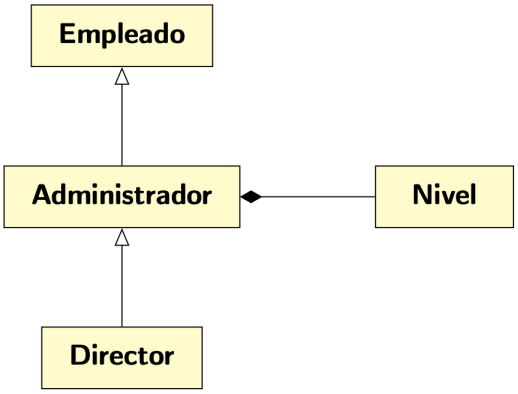
Con la *keyword* `using` hacemos que el método que especifiquemos pase de ser *privado* a *público* en la clase derivada.

```
using Clase::nombre_metodo;
```

Figura 8.5: Declaración de using

8.5. Jerarquía de Clases

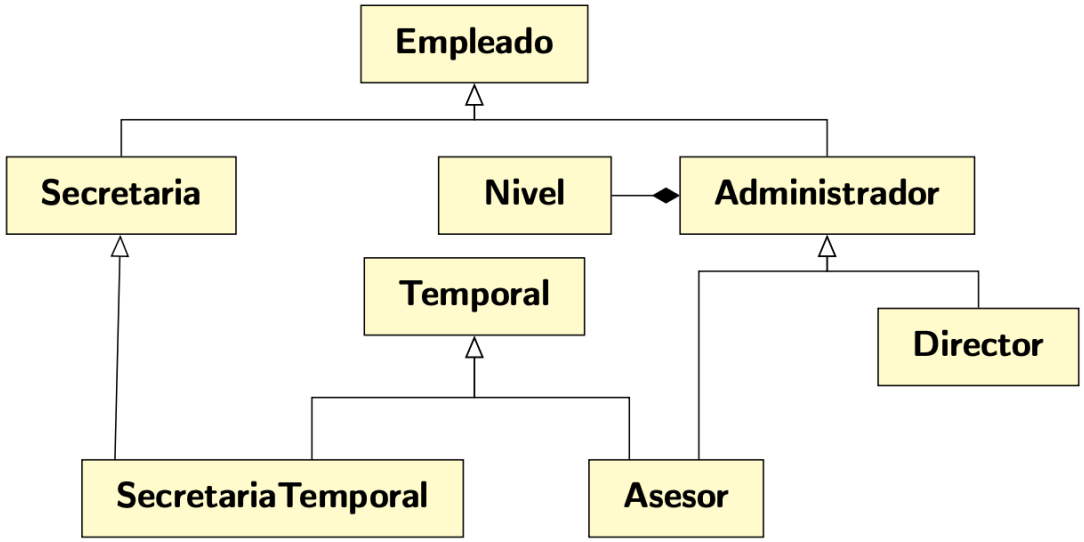
La **herencia simple** nos permite definir jerarquía entre las clases relacionadas, donde una clase se deriva solamente de una clase base, y la cadena de derivación no es circular.




```
class Empleado{//..};
class Administrador: public
   Empleado {
protected:
        Nivel nivel;
        // ...
};
class Director final: public
   Administrador { // ...};
```

Figura 8.6: Ejemplo de herencia simple

8.6. Herencia múltiple



Una clase base puede tener varias clases derivadas, además la cadena de derivación puede ser circular. Podemos crear una nueva clase combinando características de varias clases.

Vemos que la clase *Secretaria Temporal* se deriva de *Temporal* y *Secretaría*, por ejemplo. Se declara ampliando la lista de derivación.

8.6.1. Código del ejemplo anterior

```
class Temporal {//...};
class Secretaria: public Empleado {//...};
class SecretariaTemporal final: public Temporal, public Secretaria
   {//...};
class Asesor final: public Temporal, public Administrador {//...};
```

Orden de inicialización de los objetos en herencia múltiple

1. Constructores de las clases bases en el orden de la lista de derivación.
2. Constructores de los atributos en el orden declarado dentro de la clase.
3. Se ejecuta el constructor de la clase derivada.

Orden de destrucción de los objetos en herencia múltiple

1. Se llama al destructor de la clase derivada.
2. Se llama al destructor de las clases base en orden inverso al que han sido declarados.

8.7. Ambigüedades al heredar miembros sobrecargados en Herencia múltiple

Si los métodos no están en el mismo ámbito, no hay sobrecargar por tanto el compilador no sabría que método sobrecargado usar.

Cuando tenemos métodos que tienen el mismo nombre pero definidos en ámbitos diferentes, a la hora de llamarlos tenemos ambigüedades, por eso llamamos al operador de resolución de ámbito → hacemos que no haya sobrecarga.

Para que haya sobrecarga, nos llevamos todos esos métodos con el mismo nombre al mismo ámbito mediante la keyword `using`.

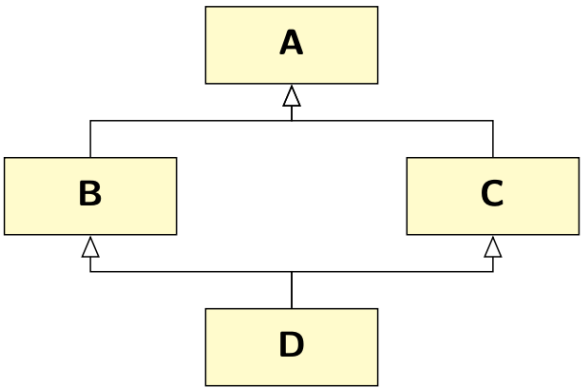
```
class B1 {
public:
  void f(char i) { std::cout << "B1::f(char)" << std::endl; }
  // ...
};

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; }
  // ...
};
```

Vemos que al hacer uso de la *keyword* `using`, declaramos como públicos los métodos `f()` de las clases bases `B1` y `B2`, resolviendo así el problema de ambigüedad a la hora de llamar a dicho método.

Programa de ejemplo:

```
int main(){
  D obj;
  // Llamadas a las funciones f para comprobar ambiguedades
  obj.f('a'); // Deberia llamar a B1::f(char)
  obj.f(10); // Deberia llamar a B2::f(int)
  obj.f(10.5);  // Deberia llamar a D::f(double)
  obj.f(10.5f); // Deberia llamar a D::f(double) debido a la
     conversion implicita
  return 0;
}
```

8.8. Herencia virtual




```
1 struct A { int a; };
2 struct B: A { int b; };
3 struct C: A { int c; };
4 struct D: B, C { int d; };
```

Figura 8.7: Problema del diamante

Para resolver el problema del diamante, podemos hacer uso de la herencia virtual, donde D tiene 2 miembros de la clase A, donde si no lo definimos así encontraríamos una ambigüedad → `d.B::a` y `d.C::a`.

```
struct A { int a; };
struct B: virtual A { int b; };
struct C: virtual A { int c; };
struct D: B, C { int d; };
```

Figura 8.8: Ejemplo de herencia virtual

Mediante la *keyword* `virtual` delante de la herencia hacemos que no haya duplicidad y D solamente tendrá un miembro `a`.

Esto hace que no haya duplicidad a la hora de acceder a los miembros de la clase A. Haciendo que la clase D tenga acceso a los miembtros: `d` (propio), `d.b`, `d.c` y `d.a` (como resultado de la herencia virtual).

8.9. Realización

Es la relación entre una *interfaz* y una clase.

Una interfaz es una clase que **solo** declara métodos no atributos, ni constructores ni destructores.

Se realiza mediante herencia, donde esa clase interfaz es una clase abstracta y es superclase.

Por ejemplo, podemos tener una interfaz `pila` y luego especializaciones de maneras de implementarla dicha pila.

Para ello, la clase que queremos que sea una interfaz / clase abstracta debe de tener como mínimo un método definido como **virtual puro**, es decir:

```
virtual tipo_metodo nombre_metodo(parametros)=0;
```

Encapsula todos los métodos que serán comunes a las clases especializadas de la misma. Tenemos que diferenciar entre **clase abstracta** e **interfaz**:

- Una **clase abstracta** es aquella que tienen al menos uno de sus métodos como **virtual puro**. Además puede tener constructores y atributos propios.
- Una **interfaz** es una clase que tiene todos sus métodos definidos como **virtuales puros**, no se puede instanciar por lo que no pueden tener constructores ni atributos propios.

9. Polimorfismo

Un método polimórfico es aquel que tiene un comportamiento diferente dependiendo del contexto en el que se realiza la llamada.

Es decir, es un método que ha sido redefinido en varias clases dependiend del uso que le queramos dar.

Tenemos un polimorfísmo en tiempo de compilación, polimorfísmo de ejecución y polimorfísmo paramétrico.

9.1. Polimorfismo en tiempo de compilación

Lo encontramos en la **sobrecarga de operadores** y en la **programación genérica** al hacer uso de *templates*.

En la sobrecarga cambia el contexto mediante la lista de parámetros que recibe el método.

- **Conversiones estándar** → Se convierte un tipo de dato a otro completamente diferente, por ejemplo de `int` a `double`.
  - Encontramos las promociones donde se convierte un tipo a uno con mayor rango (siempre del mismo tipo), por ejemplo de `int` a `long`, pero no un `int` a `double`.
- **Conversiones definidas por el usuario** → Se define mediante constructores de conversión y operadores de conversión, preferiblemente el segundo. Solo intenta una conversión implícita, si se puede realizar bien, si no lanzará un error.
- **Coincidencia con elipsis** → Si la lista de parámetros la terminamos con puntos suspensivos (...).

En las plantillas el compilador mira los tipos de los parámetros que recibe la clase.

9.2. Polimorfismo en tiempo de ejecución

```
class B {
public:
  virtual void mostrar() { std::cout << i << "␣dentro␣de␣B\n"; }
  int i;
};
class D: public B {
public:
  void mostrar() { std::cout << i << "␣dentro␣de␣D\n"; }
};
```

La clase B es una clase **polimorfica** ya que tiene un método definido como `virtual`.

La clase D tendrá dos miembros → el atributo `int i` y el método `mostrar()` , ya que hemos definido el método 'mostrar()) de la clase B como virtual, es decir el compilador sabe que ese método es polimórfico y por tanto se redefinirla en la clase derivada.

Esto sucede siempre y cuando la clase derivada tendrá un método con el mismo nombre y ese se defina como 'virtual' en la clase base.

```
int main() {
  B b, *pb = &b;
  D d;
  d.i = 1 + (b.i = 1); // b.i = 1, d.i = 1 + 1;
  pb->mostrar(); // B::mostrar()
  pb = &d;
  pb->mostrar(); // D::mostrar()
}
```

En el caso de `pb->mostrar()` ya no nos fijamos en el tipo del puntero, si no en el tipo del objeto, en este caso B.

En `pb = &d` y `pb->mostrar()` accederá al método mostrar de la clase D, ya que el objeto al que apunta es de tipo D.

Todo esto sucede ya que hemos declarado como virtual el método mostrar, si no fuera así se llamaría siempre al método que concuerde con el tipo del puntero, en este caso el de la clase B.

9.2.1. Ejemplo 1 - Polimorfismo en tiempo de ejecución

```
class B {
public:
  virtual void mostrar(int i)
    { cout << i << "␣dentro␣de␣B::mostrar(int)\n"; }
  virtual void mostrar(double d)
    { cout << d << "␣dentro␣de␣B::mostrar(double)\n"; }
  virtual void mostrar(char c)
    { cout << c << "␣dentro␣de␣B::mostrar(char)\n"; }
};

class D: public B {
public:
  virtual void mostrar(int i)
    { cout << i << "␣dentro␣de␣D::mostrar(int)\n"; }
};
```

Figura 9.1: Cabeceras

```
int main()
{
  D d;
  B b, *pb = &d;

  b.mostrar(9);      // B::mostrar(int)
  b.mostrar(9.5);    // B::mostrar(double)
  b.mostrar('a');    // B::mostrar(char)
  d.mostrar(9);      // D::mostrar(int)
  d.mostrar(9.5);    // D::mostrar(int)
  d.mostrar('a');    // D::mostrar(int)
  pb->mostrar(9);    // D::mostrar(int)
  pb->mostrar(9.5); // B::mostrar(double)
  pb->mostrar('a'); // B::mostrar(char)
}
```

Figura 9.2: Código de prueba

En este ejemplo, la clase D tiene 3 métodos dos heredados por la clase B (los que recibe un double y char) respectivamente y uno que ha sido redefinido (el que recibe un int).

Como los métodos mostrar que reciben que reciben parámetros `double` y `char` solamente se pueden acceder a ellos cuando lo llamamos mediante un objeto de la clase B.

Si hacemos `d.mostrar('a');` nos mostrará `D::mostrar(int)`, pero si hacemos `b.mostrar('a');` nos mostrará `B::mostrar(char);`

Si lo hacemos mediante punteros, se realiza un enlace dinámico que hace que comprueba el tipo del objeto de la clase, dependiendo de ese tipo llamará a un método mostrar.

Vemos que `pb` apunta a un objeto de la clase D, pero a la hora de pasarle los parámetros comprueba que método de ese objeto se corresponde a la lista de parámetros, por tanto, si se le pasa un entero y este apunta a un objeto de la clase D, se usará el método mostrar de la clase derivada D, pero como este no tiene un método redefinido para *double* o *char*, se llamará al método mostrar correspondiente del tipo del puntero, es decir, de la clase base B.

9.2.2. Ejemplo 2 - Polimorfismo en tiempo de ejecución sin interfaz

```
class Figura{
  public:
    virtual ~Figura() {};
    virtual double area()const {return 0.0;}// por omision
};
class Rectangulo: public Figura {
  public:
    Rectangulo(double lado_1, double lado_2);
    double area() const override;
  protected:
    double lado_1, lado_2;
};
class Circulo final: public Figura {
  public:
    Circulo(double radio);
    double area() const override;
  private:
    double radio;
};
```

Como vemos, la clase Figura es una clase polimorfica, de donde se derivan varias clases (Rectágunlo y Círculo).

El método `area()` es un método polimorfico que tendrá un comportamiento diferente en cada una de las clases derivadas.

> **Si hacemos el destructor virtual nos aseguramos que se llame al destructor de la clase derivada.**
> **Siempre que tengamos una clase polimorfica, su destructor debe de ser `virtual`.**

9.2.3. Ejemplo 2 - Polimorfismo en tiempo de ejecución con interfaz

```
class Figura {
public:
  virtual ~Figura();
  virtual double area() const = 0;
  // ...
  virtual void mostrar() const = 0;
};

// Destructor virtual vacío
inline Figura::~Figura() {}
```

Figura 9.3: Definición clase Figura

Convertimos la clase Figura como una clase **abstracta** haciendo que sus métodos sean *virtuales puros*.

Es decir, sus métodos solo se declaran no se implementan y además no se pueden instanciar (no se pueden crear objetos de la misma), es una clase *incompleta*.

```
class Circulo final: public Figura {
public:
  Circulo(double radio): radio(radio) {}
  double area() const override;
  void mostrar() const override;
private:
  double radio;
};
```

```
class Rectangulo: public Figura {
public:
  Rectangulo(double lado_1, double lado_2);
  double area() const final;
  void mostrar() const override;
protected:
  double lado_1, lado_2;
};
```

Encontramos la *keyword* `override` que le indicamos al compilador que dicho método se va a volver a implementar.

La *keyword* `final` indicamos que es la versión final de la implementación de dicho método.

No podemos hacer uso de `override` y `final` a la vez ya que le estamos diciendo al compilador que se va a volver a redefinir cuando no puede ya que he hemos indicado que es la última vez que se redefine con *final*.

```
class Cuadrado final: public Rectangulo {
public:
  Cuadrado(double lado);
  void mostrar() const override;
};
```

Vemos que Cuadrado es una especialización de Rectángulo pero este no redefine el método `area()` debido a que se calculan igual, por eso anteriormente la definimos como final.

9.3. Clases abstractas

Son aquellas que al menos uno de sus métodos como **virtual puro**, haciendo que dicho método se pueda volver a definir en las clases derivadas de ella.

Definición de un método virtual puro:

```
virtual void nombre_metodo() = 0;
```

Su destructor tiene que ser `virtual` para que se pueda llamar a los destructores de las clases derivadas con el fin de destruir el objeto creado correctamente (destructor de la misma clase del tipo de objeto).

Además con esto hacemos que se enlace en tiempo de ejecución debido a que no sabemos si vamos a tener o no objetos de las clases derivadas.

Haciendo que se llame primero al destructor de la clase derivada y luego al destructor de la clase base.

9.4. Operadores de conversión

Para poder realizar las conversiones correctamente entre la clase base y sus derivadas, es decir, para poder convertir un puntero de tipo base a uno de tipo derivado haremos uso de `dynamic_cast` y de `typeid()`

9.4.1. Operador de conversión: Dynamic_cast

La conversión se realiza en tiempo de ejecución a diferencia de `static_cast` que se realiza en tiempo de compilación.

Sintaxis:

`Derivada* pd = dynamic_cast<Derivada*>(pb)`
para punteros.

`Derivada& d = dynamic_cast<Derivada&>(b)`
para objetos.

Esto soluciona el peligro de realizar la conversiones explícitamente de arriba a abajo mediante `static_cast`.

Ejemplo de conversión con dynamic_cast

```
class B {
public:
  virtual ~B() {}
  // ...
};
class D: public B {
  // ...
};

void procesar(B* pb)
{
  D* pd = static_cast<D*>(pb); // Peligroso si *pb no es de tipo «D».
  if (D* pd = dynamic_cast<D*>(pb)) {
    // El objeto apuntado por «pb» es de tipo «D».
    // Así, «pb» se ha convertido sin problemas en «pd»
  }
  else { // pd == nullptr
    // El objeto apuntado por «pb» no es de tipo «D».
    // La conversión ha fallado .
  }
}
```

```
void procesar2(B& b)
{
  try {
    D& d = dynamic_cast<D&>(b);
    // El objeto «b» es de tipo «D».
    // La referencia «b» se ha convertido sin problemas en «d».
  }
  catch (std::bad_cast&) {
    // El objeto «b» no es de tipo «D».
    // La conversión ha fallado .
  }
}
```

Figura 9.4: Ejemplo de conversiones explicitas de un tipo base B a un tipo derivada D.

Vemos que comprueba si la conversión es correcta, si es así lo convierte a tipo indicado, si no, devuelve un puntero nulo (si es conversión entre punteros) o un `bad_cast` (si la conversión es entre referencias), en el caso de que sea una referencia.

Cuando trabajamos con clases polimorficas es mucho más seguro trabajar con `dynamic_cast` ya que podemos lanzar excepciones cuando la conversión no se puede realizar.

9.4.2. Operador de conversión: typeid()

Es un operador que devuelve por referencia un objeto no modificable del tipo `type_info`.

Está declarado en la cabecera `<typeinfo>`.

```
void f(Figura& r, Figura* p)
{
  typeid(r);   // tipo del objeto referido por r
  typeid(*p);  // tipo del objeto al que apunte p
  typeid(p);   // tipo Figura*; válido pero evidente
}
```

Este operador si no lo almacenamos con type_info, nos devuelve el tipo de dato al que se le pasa como parámetro.

9.5. Polimorfismo paramétrico

9.5.1. Introducción

Vamos a hacer uso de este tipo de polimorfismo mediante las plantillas de clases (**templates**).

Se realiza en tiempo de compilación.

Podemos aplicar técnicas de programación genérica, para trabajar como tipos de datos como parámetros. Podemos definir clases donde el comportamiento de los métodos/funciones de la clase genérica dependerá del tipo de dato que se pase como parámetro.

Lo importante son los requisitos que deben de cumplir esos parámetros, es decir, encontramos precondiciones para los tipos de datos de dicha plantilla. Estas plantillas pueden recibir *datos* (tipo del dato y nombre del parámetro) y *tipos de datos* (`typename` o `class`).

No usamos la plantilla como tal, si no que hacemos uso de *especializaciones* ó *instancias* de la misma.

Los métodos de una clase genérica se implementan mediante una función genérica, para ello se pone `template <typename T> Clase<T>:: nombre_metodo(parámetros)`. En este caso, incluimos el `.cpp` dentro del fichero de cabecera `.hpp` para mantener el principio de ocultación de información a la hora de instanciar plantillas.

Estas clases paramétricas las podemos relacionar con otras mediante cualquier tipo de relación vista anteriormente (asociación, composición,...).

9.5.2. Deducción de parámetro

```
template <typename T> T* crear();

void f()
{
  vector<int> v;     // el parámetro de la plantilla vector<T> es int
  int *p = crear(); // Error: imposible deducir T
  int *q = crear<int>(); // OK: T es int
}
```

El compilador mira el tipo de los parámetros reales para deducir el tipo de datos formal (T).

En algunos casos al compilador le falta información y no puede resolver el polimorfísmo paramétrico.

Vemos que en este ejemplo estamos definiendo una plantilla de función, donde el tipo de la plantilla se corresponde al tipo de los parámetros reales de la función. Si esta carece de parámetros no se podrá deducir el tipo de la plantilla.

Si el tipo de uno de los parámetros de la plantilla a función es un tipo de dato dependiente (depende del tipo de dato de la plantilla) no podemos deducir su tipo, para ello se especifica explícitamente su tipo, véase en e caso `crear<int>();`.

9.5.3. Especialización

```
template <typename T>
ostream& operator << (ostream& os, const Matriz<T>&);

template <>
ostream& operator << <bool>(ostream& os, const Matriz<bool>& M)
{
  os << boolalpha;
  for (size_t i = 0; i < filas(); ++i) {
    for (size_t j = 0; j < columnas() ++j)
      os << (*this)[i][j] << '␣';
    os << endl;
  }
  return os;
}
```

Vemos que hemos especializado el operador de inserción en flujo para un tipo determinado, en este caso un booleano.

Esta es la diferencia entre polimorfísmo paramétrico y sobrecarga de operadores.

9.5.4. Friend y Static en plantillas

Los métodos que se declaran como `friend` y no se especifica el tipo, son amigas de todas las clases, pero si a esta se le especifica el tipo, solamente será amiga a la clase que concuerde con su tipo.

Al igual sucede con los atributos que se declaran como `static`.

```
friend
Vector<T> operator *<T>(const Matriz<T>&, const Vector<T>&);
```

Solo es amiga para la clase que concuerde con el tipo T.