

# loquesiemprequisosaberdePOOynunc...



\_MDMA\_



Programación Orientada a Objetos



2º Grado en Ingeniería Informática



Escuela Superior de Ingeniería  
Universidad de Cádiz



**Descarga la APP de Wuolah.**  
Ya disponible para el móvil y la tablet.



# Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.



## LO QUE SIEMPRE QUISO SABER DE POO Y NUNCA SE ATREVIÓ A PREGUNTAR DEFINITIVE HD EDITION

### **PARTE 1: CLASES**

En la POO un programa se organiza como un conjunto finito de objetos. Los objetos de las mismas características se agrupan en **clases**, que contienen datos (**atributos**) y operaciones (**métodos**) y se comunican entre si mediante mensajes.

La estructura de una clase se divide en dos partes, pública (.h) y privada (.cpp). En la primera se define la clase, la estructura de datos mediante la que se representa y la definición de sus métodos. La parte privada contendrá la implementación de los métodos para su ocultación al usuario.

#### Parte pública:

```
#ifndef EJEMPLO_H_
#define EJEMPLO_H_

class Ejemplo{
public: //aqui las definiciones de métodos que se pueden invocar desde fuera de la clase
    Ejemplo(int n):n_(n){} //constructor de ejemplo
    void mostrar(); //muestra el entero por pantalla

private: //aqui los atributos de la clase y metodos que solo se usen dentro de la clase
    int n_; //atributo de la clase
}
```

#### Parte privada:

```
#include "ejemplo.h"
void Ejemplo::mostrar(){
    std::cout << n_ << endl;
}
```

### **PARTE 1: CONSTRUCTORES**

**Predeterminado:** se llama sin parámetros. Lo genera automáticamente el compilador, que llama a los constructores predeterminados de los miembros de la clase .

Si por ejemplo, la clase está formada por dos atributos enteros, el constructor vacío llamará a los constructores predeterminados para enteros de cada uno de los atributos. Por ello, si uno de los atributos es un puntero, no es válido el constructor del compilador y tendremos nosotros que crear uno.

```
ClaseA()
```

**Por parámetros:** recibe los parámetros necesarios para la construcción del objeto por copia.

```
ClaseA(int n){n_ = n;} //Le asignamos a n_ el valor de n.
ó
ClaseA(int n):n(n_){}  

```



WUOLAH

**De copia:** recibe una referencia constante a un objeto de la misma clase. Se llama para inicializar un objeto con otro de la misma clase y para pasar o devolver un objeto por valor. También se genera automáticamente por el compilador (a no ser que haya punteros entre los atributos).

```
ClaseA(const ClaseA& a){}
```

**De movimiento:** se llama cuando un objeto se inicializa con una referencia a otro objeto de la misma clase. Funciona igual que el de copia pero se destruye el original una vez se ha copiado el objeto.

```
ClaseA(ClaseA&& a){}
```

**De conversión:** recibe un parámetro de tipo distinto al de la clase.

```
ClaseA(const ClaseB& b){}
```

**De lista inicializadora:** recibe un objeto de tipo `initializer_list<T>`. Recibe una lista de inicializadores separados por comas {param1, param2, ...}

```
ClaseA(const std::initializer_list<T>& li){}
```

**Destructor:** si en la clase se ha creado memoria dinámica (se ha usado `new[]`) para destruirla cuando deje de usarse el objeto se utiliza `delete[]` que llama al destructor. Se llama explicitamente en raras ocasiones.

```
~ClaseA(){delete[] puntero;}
```

#### Tipos de inicialización en C++:

ClaseA a1 = v; //Sintaxis tradicional de C.

ClaseA a2 = {v}; //Inicialización de union, struct y vectores de bajo nivel.

ClaseA a3(v); //Llamada explícita al constructor

ClaseA a4 {v}; //Lista de inicialización.

## PARTE 1: OPERADORES

Pueden ser unarios (solo un operando) o binarios (dos operandos) y se pueden definir dentro o fuera de la clase. Si se definen dentro de la clase, se omite el primer operando pues el compilador presupone que este es el objeto desde el que se invoca la operación.

ClaseA **operator +**(**const** ClaseA& a); //Operador suma definido dentro de la clase.

ClaseA **operator +**(**const** ClaseA& a, **const** ClaseA& b); //Definido fuera de la clase

El operador de asignación solo se define si hay punteros en los miembros de la clase, sino vale con el por defecto del compilador.

Para diferenciar entre pre y postincremento o decremento (definidos dentro de la clase):

```
ClaseA& operator ++(); //Preincremento
```

```
ClaseA operator ++(int); //Postincremento
```

Para operadores que modifiquen el objeto (dentro de la clase) indicar que se devuelve la referencia en la declaración y, dentro de la definición devolver el puntero `*this`.

```
ClaseA& operator +=(int n); //Incremento
```

## PARTE 1: OPERADORES DE EXTRACCIÓN E INSERCIÓN DE FLUJO

El operador << se puede sobrecargar para insertar en un flujo cualquier tipo nuevo que se defina.

```
ostream& operator <<(ostream& os, const ClaseA& a){  
    os << a.atributoaimprimir;  
    return os;  
}
```

Igualmente, se puede sobrecargar el operador de extracción >> para que la clase pueda recibir valores de forma natural.

```
istream& operator >> (istream& is, ClaseA& a){  
    is >> a.atributoarellenar;  
    return is;  
}
```

Mejor definir los operadores como friend en la clase para poder acceder a sus atributos directamente.

## PARTE 1: EXCEPCIONES

Las clases de excepciones se declaran dentro de otras clases de la siguiente forma:

```
class ClaseA{  
public:  
    class Invalida{  
        public:  
            //Constructor  
            Invalida(const char* e):error(e){}  
  
            //Observador  
            const char* por_que() const{return error;}  
    private:  
        //Atributos  
        const char* error;  
    };  
};
```

Y se lanzan de la siguiente forma:

```
if(condicion de rror) throw ClaseA::Invalida("Motivo");
```

## PARTE 1: ITERADORES Y CONTENEDORES

Sirven para recorrer un contenedor. Los iteradores constantes permiten el acceso al elemento que apuntan solo de lectura. Los iteradores inversos recorren de fin a principio con el operador - - en lugar de + +.

```
typedef char* iterator;
typedef const char* const_iterator;
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

```
iterator begin(){return s_;}
iterator end(){return s_+tam_;}
const_iterator begin() const{return s_;}
const_iterator end() const{return s_+tam_;}
const_iterator cbegin() const{return s_;}
const_iterator cend() const{return s_+tam_;}
reverse_iterator rbegin(){return reverse_iterator(end());}
reverse_iterator rend(){return reverse_iterator(begin());}
const_reverse_iterator rbegin() const{return const_reverse_iterator(end());}
const_reverse_iterator rend() const{return const_reverse_iterator(begin());}
const_reverse_iterator crbegin() const{return const_reverse_iterator(cend());}
const_reverse_iterator crend() const{return const_reverse_iterator(cbegin());}
```

A continuación daremos una explicación de contenidos necesarios para la Parte 2. Existen mas, y estaría bien mirarlos (vector, list...) pero dado que son muy parecidos a sus contrapartes de bajo nivel / los que hemos hecho nosotros en AAED, id a mirarlos vosotros al cpp reference.

### **std::pair<t1,t2>**

Estructura que contiene dos elementos de tipos diferentes. Se usa en el map.

```
#include <pair>
#include <string>
#include <iostream>
int main(){
    std::pair<int,std::string> p = std::make_pair(5,"por el culo te la hinco");
    std::cout << p.first << " " << p.second >> std::endl;
}
```

### **std::set<t>**

Permite describir un conjunto ordenado y sin repetición de elementos (si se intenta añadir un elemento repetido, simplemente no ocurre).

```
#include <set>
#include <iostream>
int main(){
    std::set<int> s;
    s.insert(1); s.insert(2);
    std::set<int>::iterator i;
    for(i = s.begin(); i!=s.end();++i){
        cout << *i << endl;
}
```

# Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.



## **std::map<t1,t2>**

Permite asociar una clave a un dato. Contiene **pairs**.

```
int main(){
    map<int,string> meses;
    meses.insert(make_pair(1,"Enero"));
    //o alternativamente meses[1] = "Enero"; pero no se recomienda
    meses.insert(make_pair(2,"Febrero"));
    meses.insert(make_pair(3,"Marzo"));
    meses.insert(make_pair(4,"Abril"));
    meses.insert(make_pair(5,"Mayo"));
    //etc...
    map<int,string>::const_iterator i = meses.find(3);
    cout << "El mes " << i->first << " es " << i->second << endl;
    //o alternativamente cout << "El mes 1 es " << meses[1]<< endl; pero no se recom.

    for(i=meses.begin();i!=meses.end();i++)
        cout << "El mes " << i->first << " es " << i->second << endl;
}
```

## **PARTE 1: Funciones, consejos, recomendaciones...**

Si declaras una función o clase como **friend** o amiga de otra clase, esta tendrá acceso a sus elementos privados.

Si se declara un miembro como **static** o estático este pertenecerá a la clase, y no al objeto. Es decir, si lo cambiamos, lo cambiamos para todos los objetos instanciados de esa clase. Se llama al atributo mediante el operador de resolución de ámbito Clase::atributoestático.

Si un atributo se declara como **mutable** puede ser modificado incluso desde métodos constantes.

**const** al principio de una función significa que esta devuelve un objeto constante. Al final de la función significa que el **método es constante**, es decir, no modificará ningún atributo de la clase que no sea **mutable**.

### **Extraer la fecha del sistema y guardarla en dia, mes, año:**

```
#include<ctime>
time_t tmp_syst = time(nullptr);           //Obtiene la fecha del sistema
tm* tmp_partes = localtime(&tmp_syst); //Divide en campos la fecha.
dd_ = tmp_partes -> tm_mday;
mm_ = tmp_partes -> tm_mon+1;
aa_ = tmp_partes -> tm_year+1900;
```

**Recomendación:** definir los operadores unarios dentro de la clase y los binarios fuera de ella.

Para ejercicios de ejemplo descargar: *RecopilacionEjerciciosParte1.pdf*



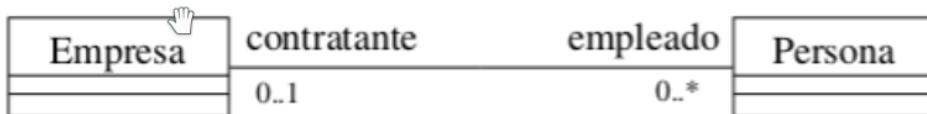
**WUOLAH**

## PARTE 2: ASOCIACIONES

Un enlace representa una conexión entre objetos y asociación una entre clases (son una abstracción de las primeras). Propiedades de una asociación:

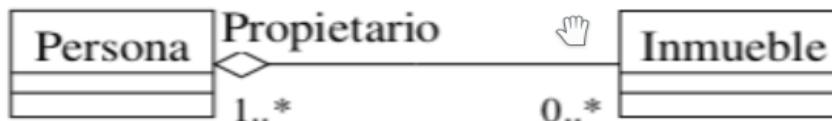
- **Cardinalidad:** nº de clases que intervienen en la asociación.
- **Multiplicidad:** número de objetos que puede haber en cada extremo de la asociación (1:1, 1:n, m:n...).
- **Navegabilidad:** sentido de la asociación (unidireccionales o bidireccionales).

*<Trabaja para*



## PARTE 2: AGREGACIONES

Asociación asimétrica donde existe una relación del tipo “todo/parte” o “está compuesto por”, donde objetos que representan componentes de algo se enlazan a otro que representa un ensamblaje completo.



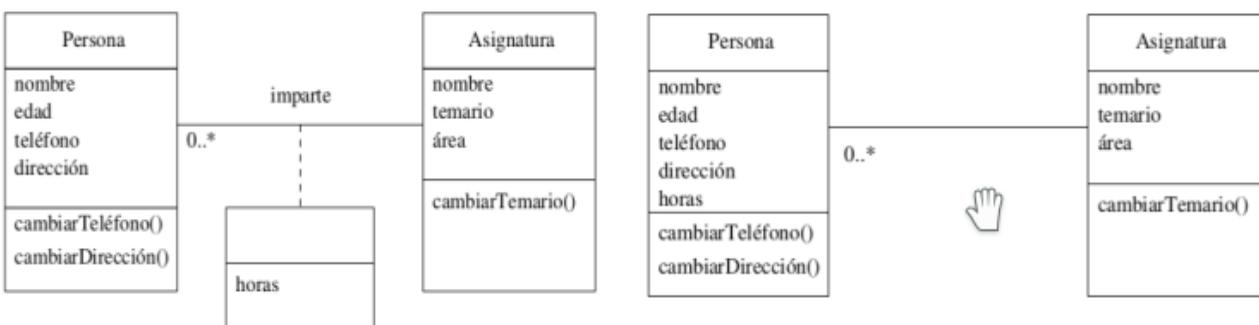
Las **composiciones** constituyen un caso especial de la agregación donde los componentes están físicamente contenidos por el agregado. Implica que la multiplicidad en el lado del agregado solo puede ser 0..1.



Las agregaciones son transitivas (si C pertenece a B y B pertenece a A entonces C pertenece a A) y asimétricas (si B pertenece a A entonces A no puede pertenecer a B).

## PARTE 2: ATRIBUTOS DE ENLACE

Atributos que pertenecen a la asociación y no a ninguna de las dos clases en particular. Es posible pasar los enlaces a una de las clases en las asociaciones uno a uno y uno a varios, poniéndolo en la clase que está al otro lado del uno (**!!** solo cuando tenga sentido). No se puede hacer en las asociaciones varios a varios.



## PARTE 2: CALIFICACIÓN

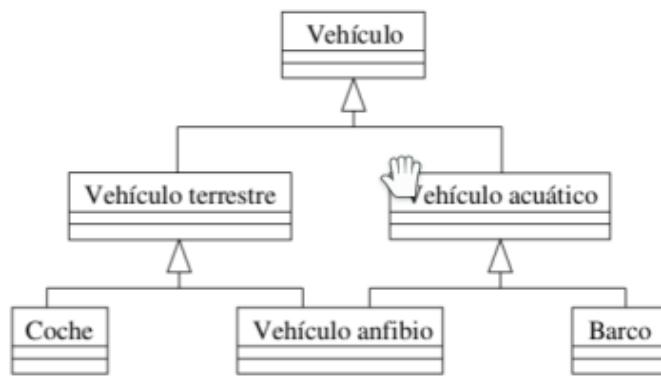
Solo aparecen en las asociaciones con multiplicidad varios en uno de sus extremos, y solo puede aparecer en ese extremo. Permite distinguir objetos individuales en ese extremo. Es un atributo especial que reduce la multiplicidad efectiva de la relación.

## PARTE 2: GENERALIZACIÓN Y ESPECIALIZACIÓN

**Especialización:** relación entre una clase y una o mas versiones especializadas de la misma. En el sentido inverso se llama **generalización**.

Los atributos y operadores de la clase o madre o superclase se propagan a las subclases. Una subclase no puede suprimir un atributo ni una operación, aunque si puede reescribir estas últimas (**redefinición**). Tambien pueden (y deben) añadir nuevas características (**extensión**).

El mecanismo mediante el que se implementa la generalización se llama **herencia**. Esta puede ser simple (cada subclase una superclase) o múltiple (una subclase hereda de multiples superclases).



## PARTE 2: CLASE ABSTRACTA Y REALIZACIÓN

Una **clase abstracta** es una clase de la que no se pueden crear objetos por ser alguna de sus operaciones abstractas (solo se especifican, no se definen). Una subclase no abstracta de una clase abstracta deberá definir todos los métodos abstractos de su madre.

Una clase abstracta sin atributos y contodos sus métodos abstractos se denomina **interfaz**. Una relación entre una interfaz y una clase se llama **realización**.

## PARTE 2: IMPLEMENTACIÓN DE ASOCIACIONES

Los metodos y atributos necesarios para implementar asociaciones se incluyen en las definiciones de las clases implicadas.

### Asociación Binaria:

Se implementa normalmente con dos miembros de datos, uno para cada una de las clases asociadas. En caso de que en un extremo la multiplicidad sea uno el miembro de datos que le corresponde será un punter a un objeto de la otra clase.

```

Asoc. Binaria 1-1
class A{
public:
    void relaciona(B& b);
    B& relaciona() const;
private:
    B* b;
}

class B{
public:
    void relaciona(A& a);
    A& relaciona() const;
private:
    A* a;
}

```

Para relacionar una instancia de B con una de A:

```

void A::relaciona(B& b){
    this → b = &b;
}

```

Para ver que instancia de B está relacionada con una de A:

```

B& A::relaciona() const{
    return *b;
}

```

Para recorrer la relación:

```

if(!b) //Comprobar que exista la relación
else b → "atributo/metodo de b"

```

# Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.



## Asoc. Binaria Varios – Varios

Cuando uno de los extremos es de multiplicidad varios le corresponde **set**. El SET es un contenedor en el que se insertan elementos únicos (no se puede almacenar dos elementos iguales) y en un orden concreto. Una vez dentro no se pueden modificar hasta que se saquen del mismo.

```
#include <set>
class A{
public:
    void relaciona(B& b);
    const Bs& relaciona() const;
private:
    typedef set<B*> Bs;
    Bs bs;
}

class B{
public:
    void relaciona(A& a);
    const Bs& relaciona() const;
private:
    typedef set<A*> As;
    As as;
}
```

Para relacionar una instancia de B con una instancia de A

```
void relaciona(B& b){
    bs.insert(&b);
}
```

Para obtener las instancias de B relacionados con una instancia de A:

```
const Bs& asocia() const{
    return bs;
}
```

Para recorrer la relación:

```
if(bs.empty()) //Comprobar que existe la relación
for(A::iterator i= bs.begin(); i!=bs.end(); i++)
    (*i) → metodo();
```

Si la asociación es unidireccional solo hará falta definir los atributos en la clase de origen.



### **Asoc. Calificadas en un extremo**

Para implementar las asociaciones calificadas usaremos **map**. Es un contenedor **map<key,elemento>** donde se asocia un elemento a una determinada llave. Suelen ser mas lentos que los **unordered\_map** pero permiten el acceso directo a los elementos asociados mediante su llave usando el operador [].

```
#include <map>
class A{
public:
    void asocia(B&);
    const BsCalificadas& asocia() const;
private:
    typedef map<C,B*> BsCalificadas;
    BsCalificadas bs;
}

class B{
public:
    void asocia(A&);
    const As& asocia() const;
private:
    C c; //calificador
    As as;
}

void A::asocia(B& b){
    bs.insert(make_pair(b.c(),&b));
}

const A::BsCalificadas& A::asocia() const{ return bs;}
```

Para recorrer la relación:

```
if(bs.empty())
else
for(A::BsCalificadas::const_iterator i = bs.begin(); i != bs.end(); ++i)
    (i->second)→metodo();
```

### **Asoc con atributos de enlace**

Si A está relacionado con B y hay un atributo 'at' de relación se hace un map, en lugar de entre A y B, entre B y at en A y entre A y at en B. En algunos casos se puede meter el atributo dentro de una de las dos clases en la asociación.

## PARTE 2: CLASES DE ASOCIACIÓN

En lugar de implementar las asociaciones mediante atributos en las clases que participan en la misma se puede crear una clase de asociación explícita, para lo que usaremos tanto SET como MAP.

```
#include <map>
#include <set>
class AsociacionBidireccional{
public:
    void asocia(A& a, B& b);
    void asocia(B& b, A& a);
    std::set<B*> asociados(A& a) const;
    std::set<A*> asociados(B& b) const;
private:
    std::map<A*,std::set<B*> > directa;
    std::map<B*,std::set<A*> > inversa;
}
```

Para establecer la relación:

```
void AsociacionBidireccional<A,B>::asocia(A& a, B& b){
    directa[&a].insert(&b);
    inversa[&b].insert(&a);
} //Para la inversa, llamar a asocia cambiando los parámetros de orden.
```

Devuelve el conjunto de enlaces asociados a un objeto .

```
std::set<B*> AsociacionBidireccional<A,B>::asociados(A& a) const{
    typename std::map<A*, std::set<B*> >::const_iterator i = directa.find(&a);
    if (i != directa.end()) return i->second;
    else return std::set<B*>();
}
```

## Clases de asociación con atributos de enlace

```
#include <map>
using std::map;
using std::make_pair;
// A y B: clases asociadas
// Z: clase de los atributos de enlace
class AsociacionBidireccional {
public:
    void asocia(A& a, B& b, Z& z);
    void asocia(B& b, A& a, Z& z);
    map<B*, Z*> asociados(A& a) const;
    map<A*, Z*> asociados(B& b) const;
private:
    map<A*, map<B*, Z*> > directa;
    map<B*, map<A*, Z*> > inversa;
};
```

Para establecer la relación

```
void AsociacionBidireccional<A, B, Z>::asocia(A& a, B& b, Z& z){  
    directa[&a].insert(make_pair(&b, &z));  
    inversa[&b].insert(make_pair(&b, &z));  
}
```

Recorrer la relación

```
map<B*,Z*> AsociacionBidireccional<A, B, Z>::asociados(A& a) const{  
    map<A*, map<B*, Z*>>::const_iterator i = directa.find(&a);  
    if (i != directa.end()) return i->second;  
    else return map<B*, Z*>();  
}
```

### Implementación de agregaciones

Se implementan igual que las asociaciones. En caso de ser una composición, el componente puede modelarse como un atributo normal del agregado.

### Implementación de generalizaciones

Se implementan mediante herencia

```
class clasederivada: [accesibilidad] clase-base1, [accesibilidad] clase-base2...{  
    //miembros  
};
```

Se heredan todos los miembros menos constructores, destructores y operadores de asignación. La accesibilidad por defecto es private.

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

Se puede usar el constructor de la clase base en el constructor de la clase derivada de la siguiente forma:

```
class A{  
public:  
    A(int k, char c):k(k),c(c){}  
private:  
    int k;  
    char c;  
}
```

# Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.



```
class B: A{  
public:  
    B(int k, char c, bool t): A(k,c),t(t){}  
private:  
    bool t;  
}
```

**!!** En caso de herencia múltiple, respetar el orden de los constructores

## PARTE 2: COSAS A TENER EN CUENTA

Resolución de ámbito: si B hereda de A y tiene un método homónimo a A, para invocar el método de A desde B es necesario el operador de resolución de ámbito.  
(Ej: b.A::método())

```
#include "estudiante.h"  
#include "doctorando.h"  
  
int main()  
{  
    Estudiante e("María Pérez Sánchez", 31682034);  
    Doctorando d("José López González", 32456790,  
                 "Dr. Juan Jiménez", 134);  
  
    Estudiante* pe = &e;  
    pe->mostrar(); // e.mostrar()  
    pe = &d; // conversión «hacia arriba»  
    pe->mostrar(); // d.estudiante::mostrar()  
    Doctorando* pd = &d;  
    pd->mostrar(); // d.mostrar()  
    pd->Estudiante::mostrar(); // d.estudiante::mostrar()  
}
```

Los objetos de tipos relacionados por **herencia pública** son compatibles hacia arriba, un objeto de A se puede inicializar con un objeto de B (en el ejemplo de constructores, se perdería el atributo bool t). **NO FUNCIONA VIVEVERSA.**

```
#include "estudiante.h"  
#include "doctorando.h"  
  
int main()  
{  
    Estudiante e("María Pérez Sánchez", 31682034), *pe;  
    Doctorando d("José López González", 32456790,  
                 "Dr. Juan Jiménez", 134), *pd;  
  
    pe = &d; // bien  
    pd = pe; // ERROR  
    pd = static_cast<Doctorando*>(pe); // bien  
    e = d; // bien  
    d = e; // ERROR  
    d = Doctorando(e); // ERROR  
    d = static_cast<Doctorando>(e); // ERROR  
    d = reinterpret_cast<Doctorando>(e); // ERROR  
}
```



WUOLAH

Si una clase D hereda públicamente de B, se puede asignar un D\* a un B\* sin usar conversión explícita de tipos. La conversión inversa, de B\* a D\*, tiene que ser explícita ya que, en principio, no es segura.

**Orden de inicialización:** Si D hereda de B y C y estos a su vez de A, primero se inicializa el objeto A, luego B y C (según su orden en la lista de herencia), luego los miembros de D y por último el objeto D. Los destructores se ejecutan en orden inverso.

**Ambigüedades:** si una clase derivada hereda miembros que se llamen igual de dos clases base distintas, hay que usar el operador de ámbito para distinguirlos entre ellos, aunque sean funciones del mismo nombre que reciban parámetros distintos (sobrecarga).

Si se quiere que haya sobrecarga de operaciones, declarar las funciones en la clase derivada de la siguiente forma:

```
using B1::f();  
using B2::f();
```

Si una clase D hereda de B y C, que a su vez heredan de A, para evitar que los atributos de A se hereden por duplicado esta tiene que ser heredada de forma virtual:

```
class B: virtual A{}  
class C: virtual A{}
```

Para ejercicios de ejemplo descargar: *RecopilacionEjerciciosParte2.pdf*

### PARTE 3: SOBRECARGA

Si a estas alturas todavía no sabes que si dos funciones que se llaman igual el compilador diferencia a cual tiene que llamar según los parámetros que recibe lo llevas chungo chav@l. Esto es **polimorfismo en tiempo de compilación**.

### PARTE 3: FUNCIONES MIEMBRO VIRTUALES

La palabra reservada virtual es un especificador de función que se aplica solo a declaraciones de métodos para conseguir **polimorfismo en tiempo de ejecución**.

Un puntero de la clase base puede apuntar a un objeto tanto de la clase base como de la derivada. Si se llama a un método a través de tal puntero, la selección dependerá del tipo del puntero, no obstante, si el método se ha definido como virtual, la selección dependerá del objeto al que apunte. En caso de que no exista este método en la clase derivada se llamará al virtual de la clase base.

Una vez declarado virtual esta propiedad se propaga a todas las redifiniciones de las clases derivadas. Es decir, cuando se redefine un método virtual en la clase derivada, aunque no le pongas virtual, será virtual.

Llamar a una función utilizando el operador de resolución de ámbito aseguro que no se emplea el mecanismo virtual.

Los constructores no pueden ser virtuales, pero si los destructores.

Una clase que tenga métodos virtuales debe tener un destructor virtual, aunque esté vacío.

### PARTE 3: CLASES ABSTRACTAS

Son clases que contienen un método virtual puro, donde ni siquiera se define el cuerpo.

**virtual** prototipo() = 0;

Viene a ser como decir "este método ya se definirá en otro sitio (en sus clases derivadas, específicamente).

### PARTE 3: IDENTIFICACION DE TIPOS EN TIEMPO DE EJECUCIÓN

C++ posee mecanismos para determinar con seguridad a partir de un puntero o referencia al tipo real del objeto al que se hace referencia.

**dynamic\_cast<tipo>(valor)** donde tipo es un tipo puntero o referencia y valor un valor apropiado para su conversión a dicho tipo.

Si la conversión es válida se devuelve el valor del tipo requerido, sino, se devuelve 0 si se quiere convertir a un puntero o se lanza la excepción **bad\_cast** si el tipo destino era una referencia.

```
void proc_esar(B* pb){  
    if (D* pd = dynamic_cast<D*>(pb))  
        // El objeto apuntado por pb es de tipo D, haz noseque  
    else  
        // El objeto apuntado por pb no es de tipo D haz otra cosa  
}
```

Incluido en la cabecera <typeinfo> existe también el operador:

**typeid**(objeto/referencia/puntero/nombredetipo)

que devuelve una referencia a un objeto global no modificable de tipo type\_info. Estos objetos se pueden comparar mediante == y != y se puede obtener el nombre mediante el método name() o clasificar mediante before().

### **PARTE 3: PLANTILLAS Y POLIMORFISMO PARAMÉTRICO**

Visto en AAED podemos hacer una clase general que se especialice automáticamente según el tipo de datos que va a contener. Para ello incluir:

**template <typename T> class Algo{};**

Para definir los métodos de la plantilla se pondrá el tipo como Algo<T>...

Al crear objetos de esta clase habrá que especificar su tipo. Algo<int> o Algo<char> etc...

Las clases paramétricas pueden tener **clases y funciones amigas**. Si la función amiga no utiliza plantillas es universal a todas las especializaciones de la clase paramétrica. Si una función amiga usa plantillas es específica de su clase especializada.

Una clase paramétrica puede poseer **miembros estáticos**, siendo éstos específicos de cada especialización.

Una plantilla puede estar compuesta por varios parámetros:

**template <typename T1, typename T2>**

Una plantilla puede tener parámetros de un tipo pre-existente

**template <typename T, size\_t n>** a la que se llamaría como Algo<char, 20> por ejemplo.

También se le pueden proporcionar valores por omisión.

**template <typename T = char, size\_t n = 256>**

Para ejercicios de ejemplo descargar: *RecopilacionEjerciciosParte3.pdf*