

Programación Orientada a Objetos con C++

Juan A. Romero - aromero@uco.es

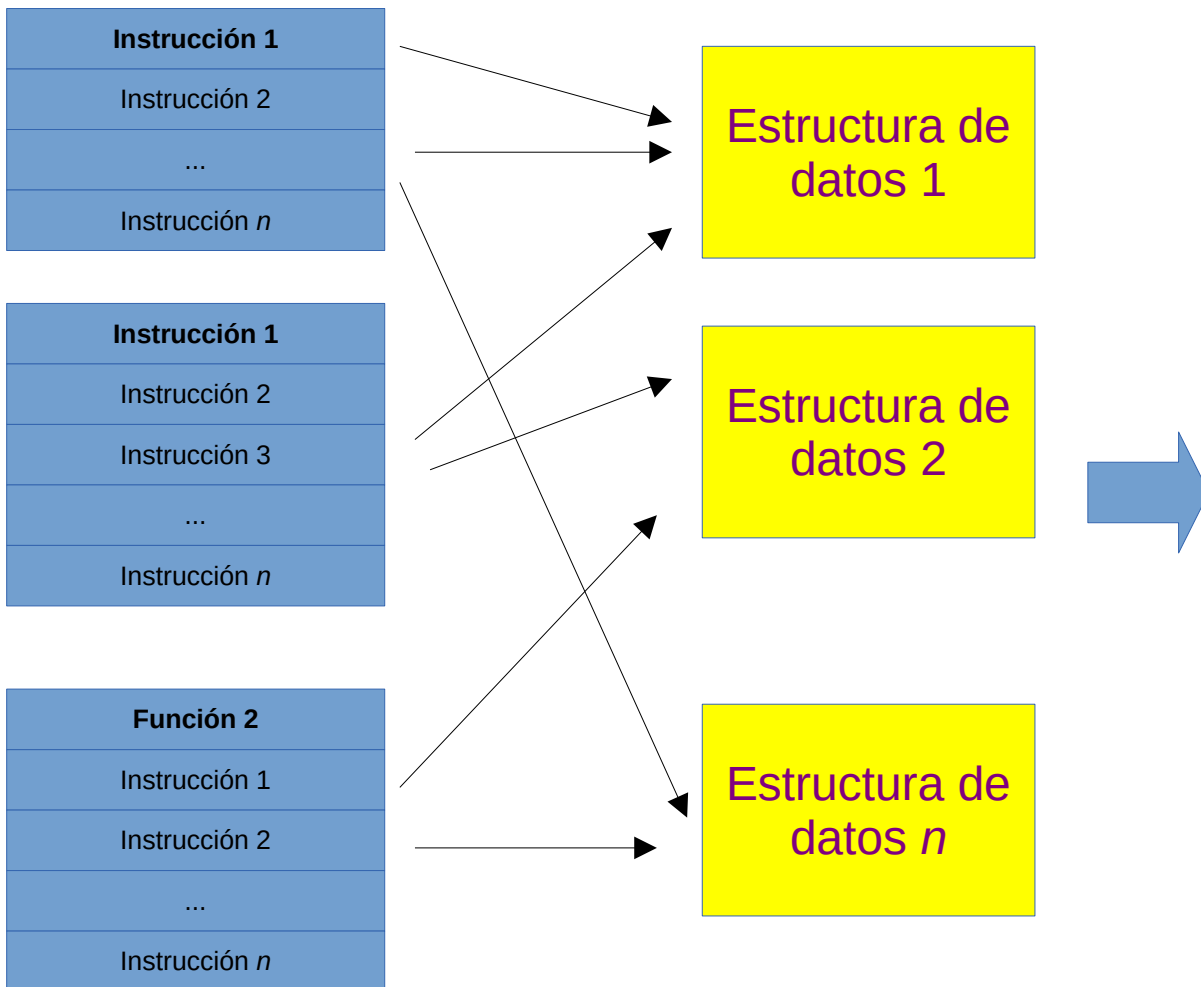
25 septiembre 2024

**Programación/Diseño Orientado a
Objetos (POO/DPO)**

**Object Oriented Programming/Design
(OOP/OOD)**

¿Qué es la POO? ¿Qué es un 'objeto'?

- Programando sistemas complejos...



¿Pensar todo en término de Estructuras de Datos e instrucciones?

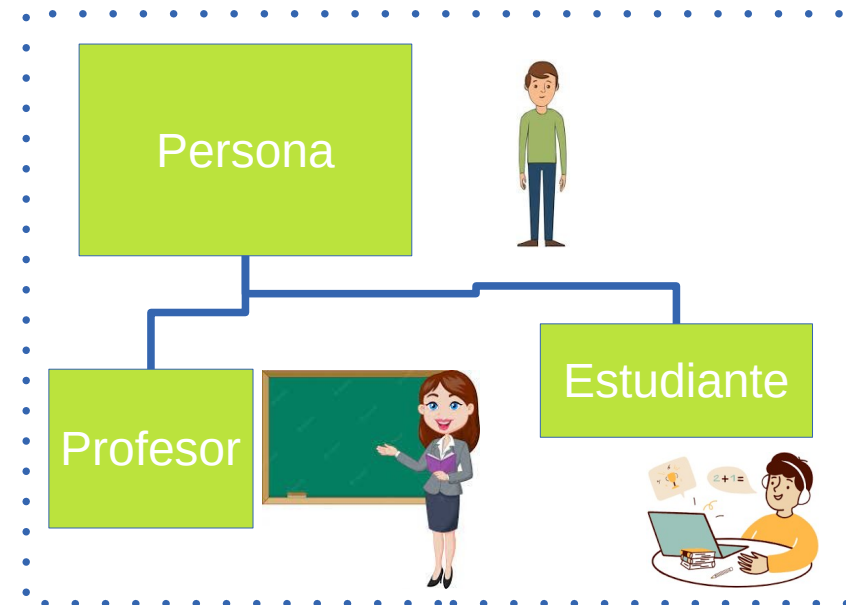
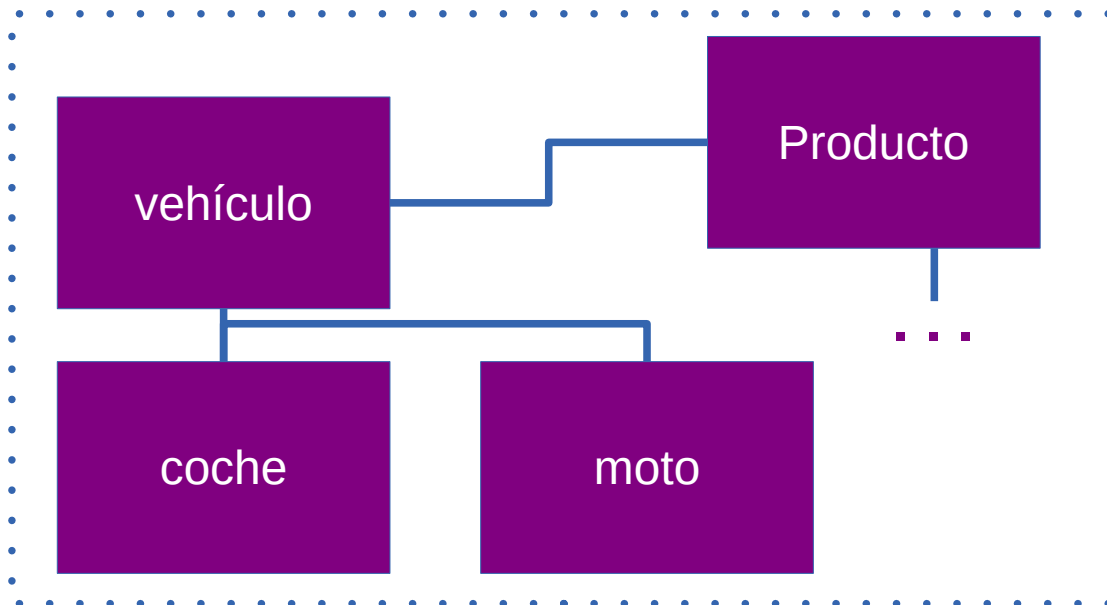


¿Qué es la POO? ¿Qué es un 'objeto'?

- Programando sistemas complejos...

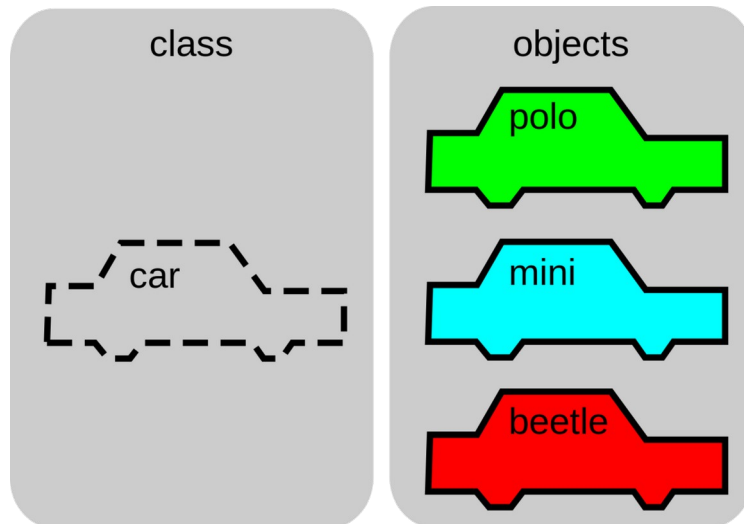


*Es mucho más natural,
sencillo y potente:
¡Pensar todo en término
de **objetos**!*



¿Qué es la POO? ¿Qué es un 'objeto'?

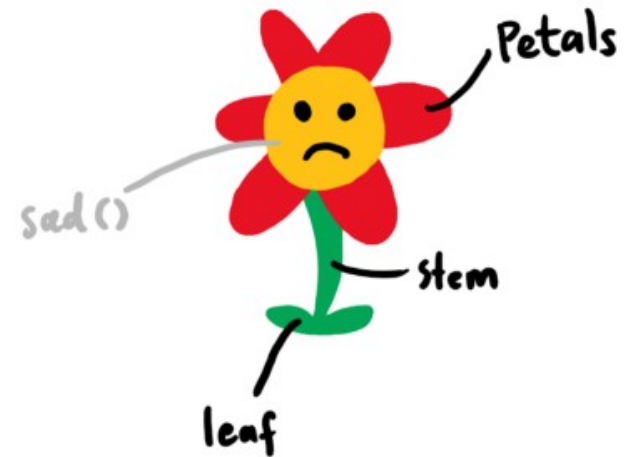
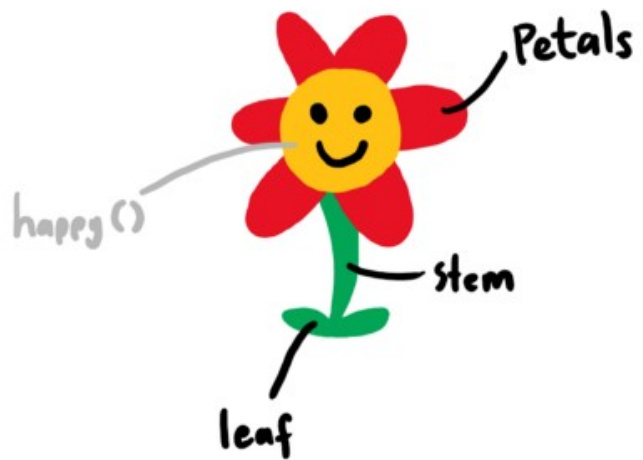
- ¿Vendría bien una entidad 'coche' en mi programa?
- ¿Qué puede hacer el objeto 'coche' en mi programa?
- ¿Cómo puedo construir un objeto 'coche?', ¿qué datos y acciones necesita?
- ¿Hay alguna relación entre 'coche' y 'furgoneta'? ¿entre coche y una lista de vehículos?



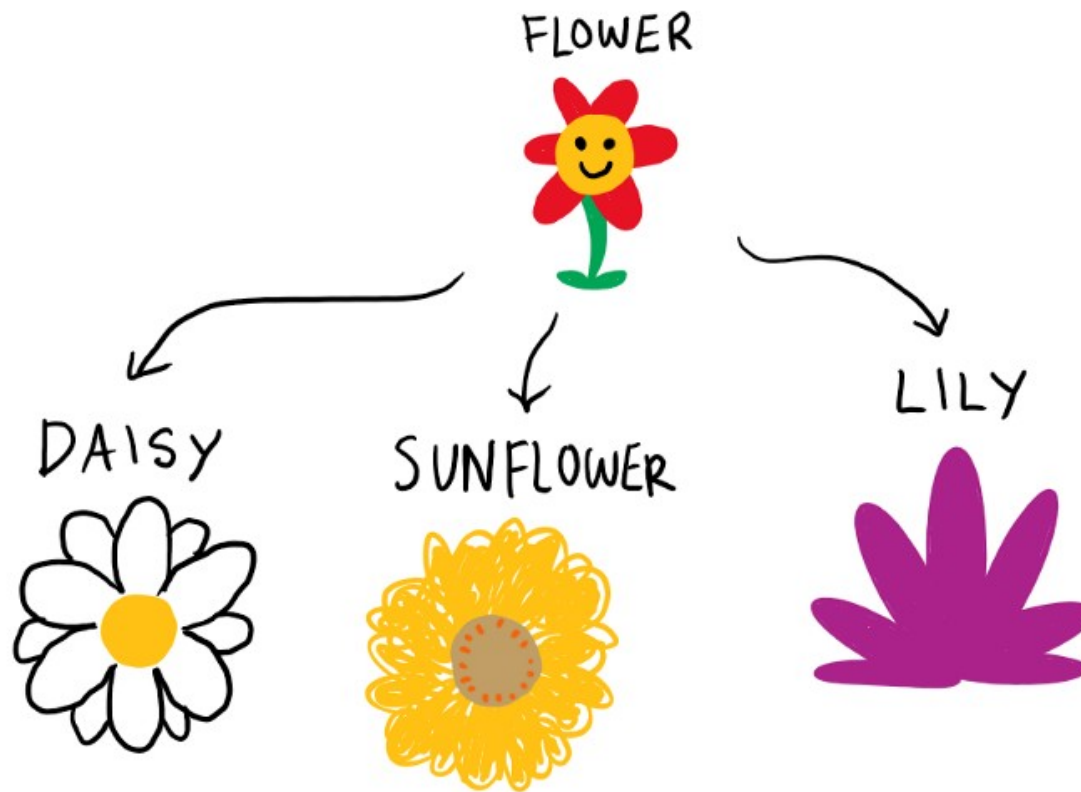
- Resulta que en un programa/sistema (y en el mundo real) que esté diseñando/construyendo: **todo son objetos**.

¿Y una clase?

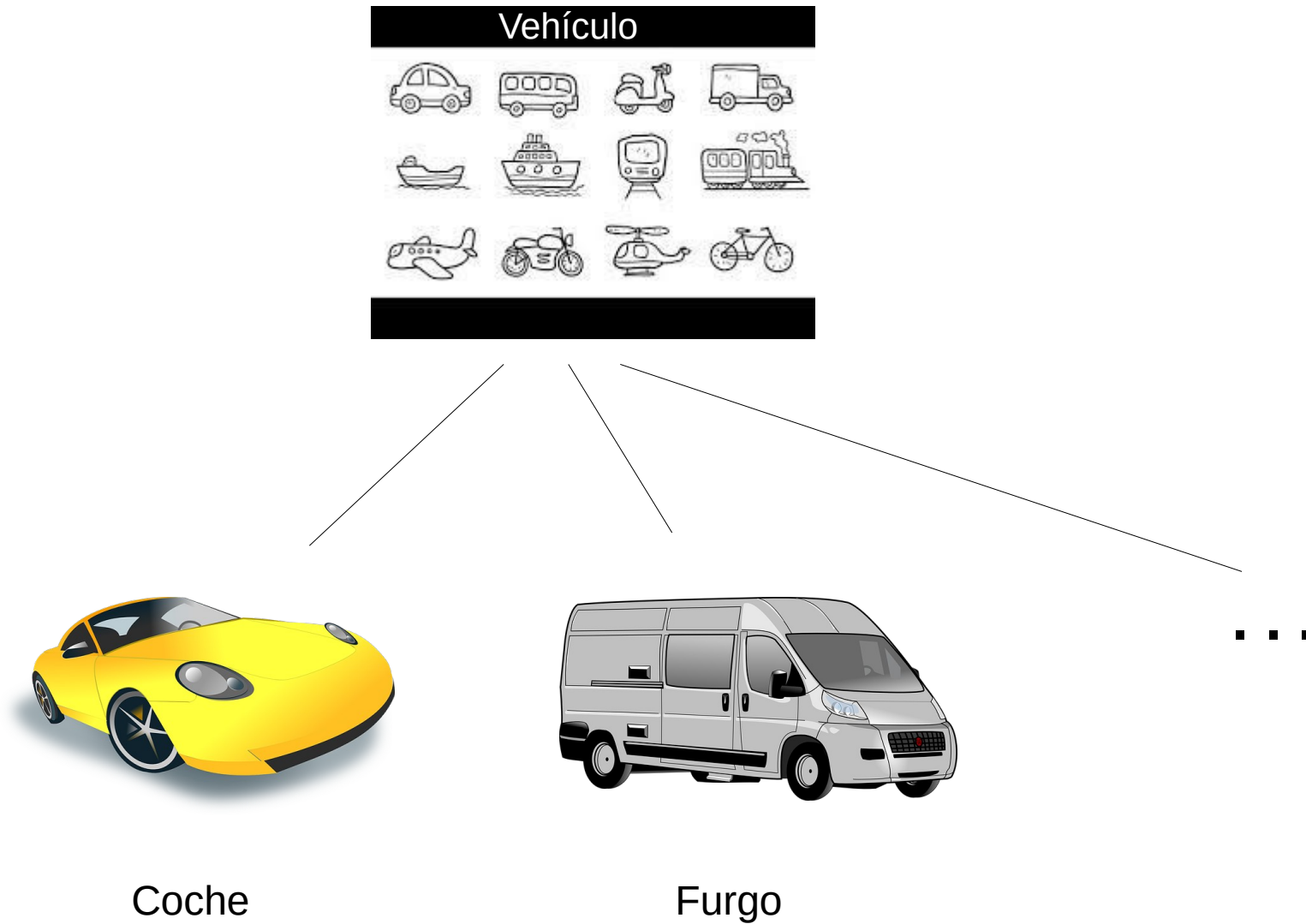
class Flower:



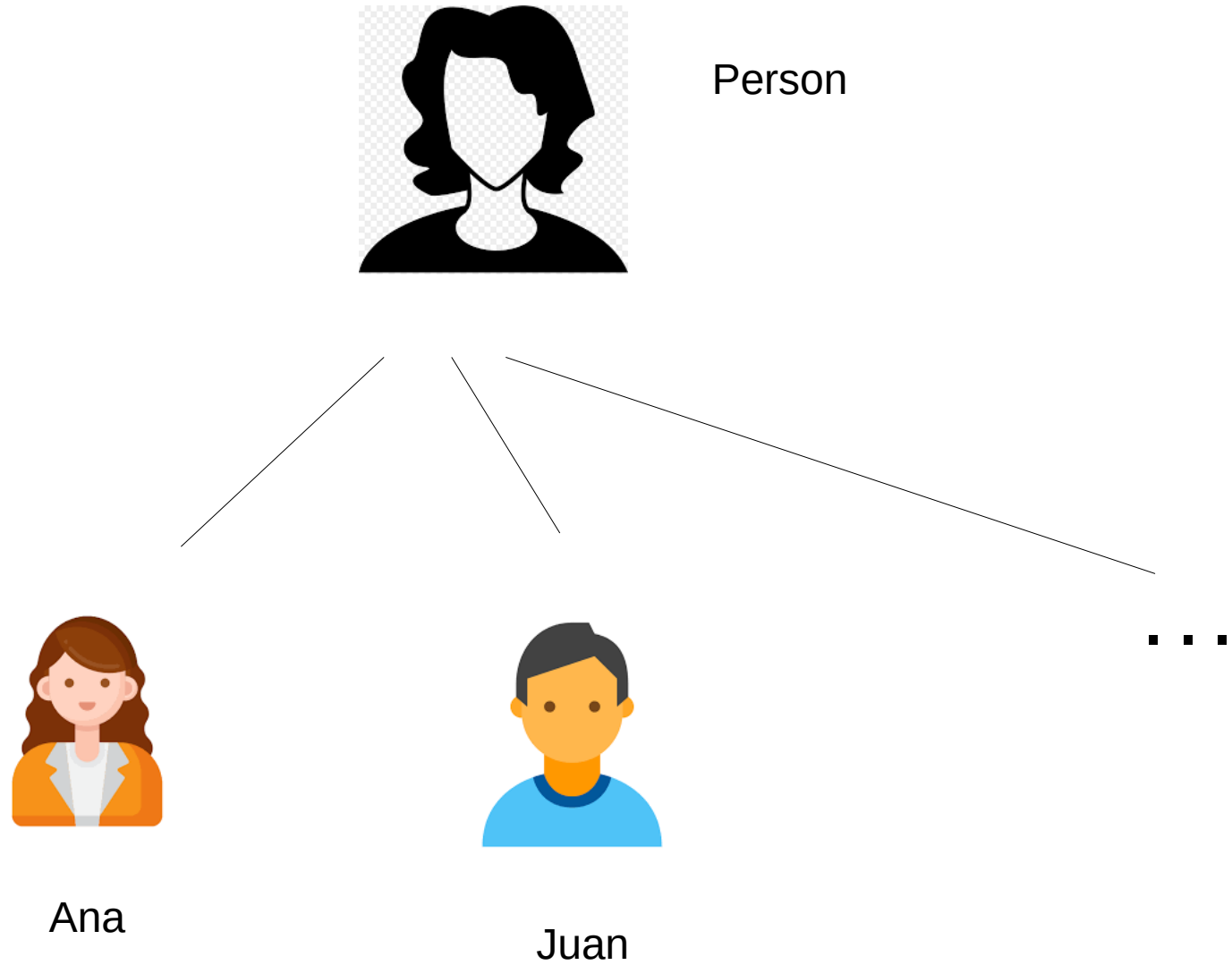
superclass/subclass



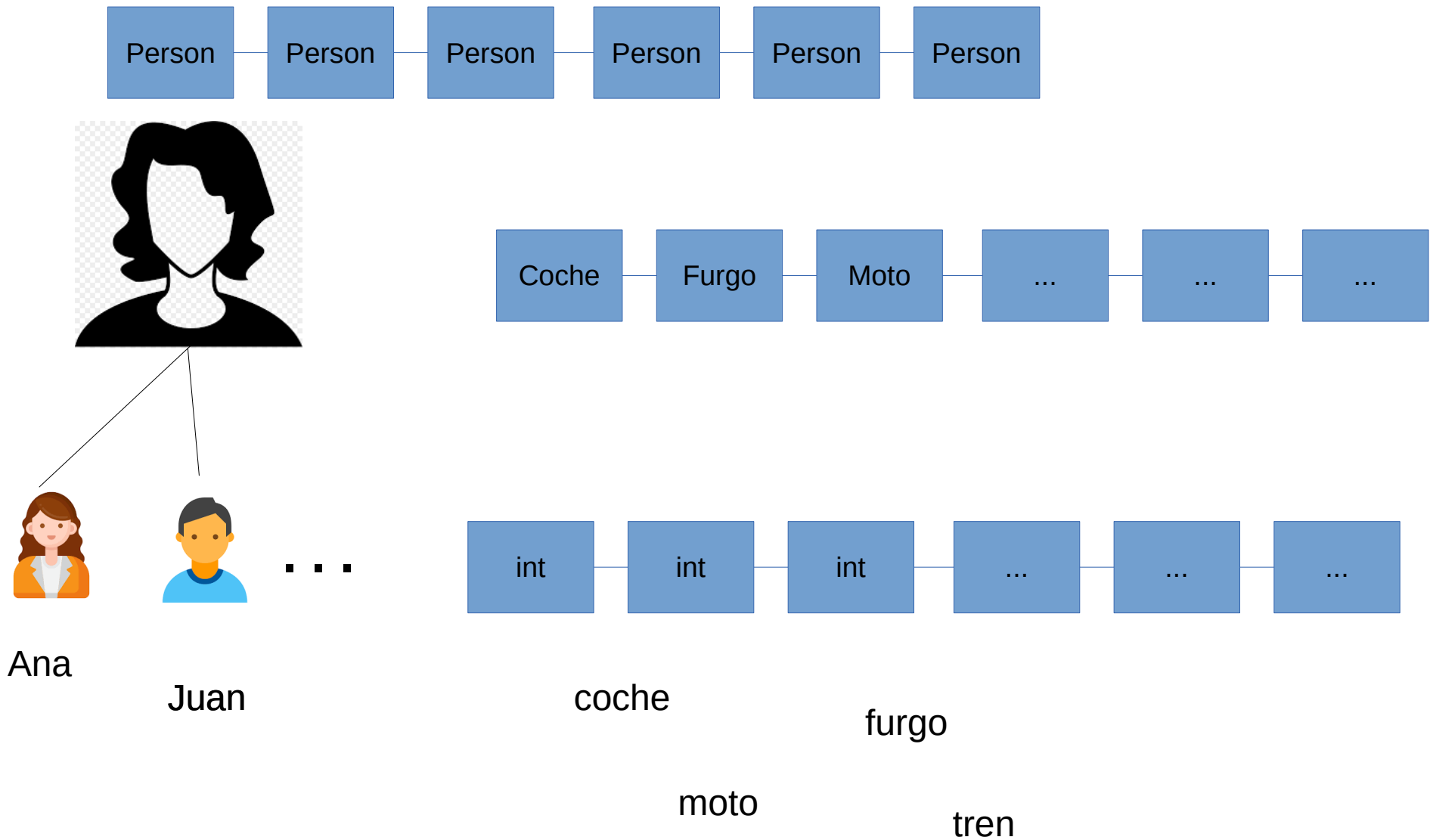
superclass/subclass



superclass/subclass



Relaciones entre clases



¿De dónde salen las clases en mi programa?

- Cuando mi programa necesita representar una entidad del mundo real, como una persona, un coche, una factura, una lista o un archivo, etc.
- Cuando mi programa necesita representar una entidad que no existe en el mundo real (ficticia) pero que yo la creo útil para hacer un programa. Como un menú de una aplicación, un botón, una tabla de datos, un buscador, etc.
- » Represento sus datos y su comportamiento dentro de mi programa mediante la creación de una clase. (ESTO ES UNA DEFINICIÓN DE 'CLASE').
- » Una clase es un modelo, una plantilla, un patrón general. Con esa plantilla (o clase) puedo crear 'objetos' concretos (instancias) de esa clase. (ESTO ES UNA DEFINICIÓN DE 'OBJETO').
- Instancias u objetos de la clase 'Person' son, por ejemplo: Juan, Ana y Raquel.

Componentes de una clase: datos + métodos (funciones)

Button
- xsize - ysize - label_text - interested_listeners - xposition - yposition
+ draw() + press() + register_callback() + unregister_callback()

**Diagrama de clases
(UML)**

y ... ¿cómo programamos con POO?

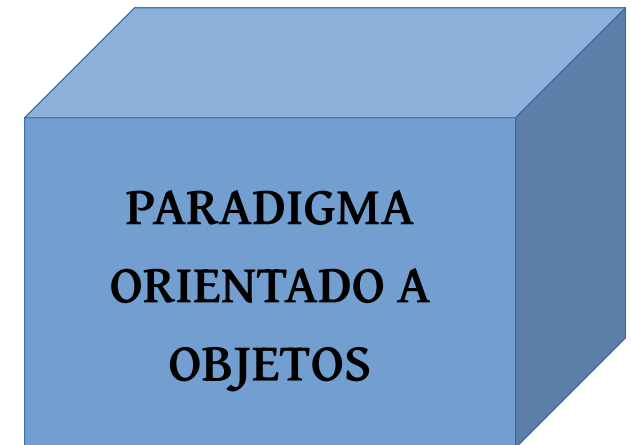
- Análisis (del problema e identificación de las clases)
- Diseño (Orientado a Objetos) de las clases surgidas del análisis y de nuevas clases que puedan surgir en el diseño
- Implementación en un LPOO (C++, Python, Java, etc.)

En C++

```
class Person{  
private:  
    string name_;  
    int age_;  
    . . .  
public:  
    void SetName(string name);  
    . . .  
}
```

DATA

**FUNCTION/
BEHAVIOUR**





official isocpp logo

Práctica 1... (la práctica 1 es autoexplicativa ,y más importante que hacer el código es entender bien las explicaciones de todos los conceptos fundamentales que trata)

Breve Historia

- *Bjarne Stroustrup* (Dinamarca) en 1979 desarrolla C++ en los laboratorios *Bell Labs* (EEUU).
- Se trata de un “C” con clases. Por ello es considerado a veces un LPOO “no *puro*”...
- ISO comienza estandarización desde 1998 hasta hoy: C++98, C++03, C++11, C++14, C++17 (isocpp.org es fundamental para su desarrollo y su éxito).
- The current ISO C++ standard is **C++20**.
- In-progress **C++23**.
- Referencias importantes/fundamentales con las que debemos familiarizarnos:
 - *Standard C++ Foundation* <https://isocpp.org/about>
 - *cppreference*. <https://cppreference.com>
 - *cplusplus.com*. <https://cplusplus.com>
 - *stackoverflow.com*. <https://stackoverflow.com>
 - *GitHub*. <https://github.com>

Styleguides

Guía de estilo de Google

- <https://google.github.io/styleguide/cppguide.html>

Hay otras:

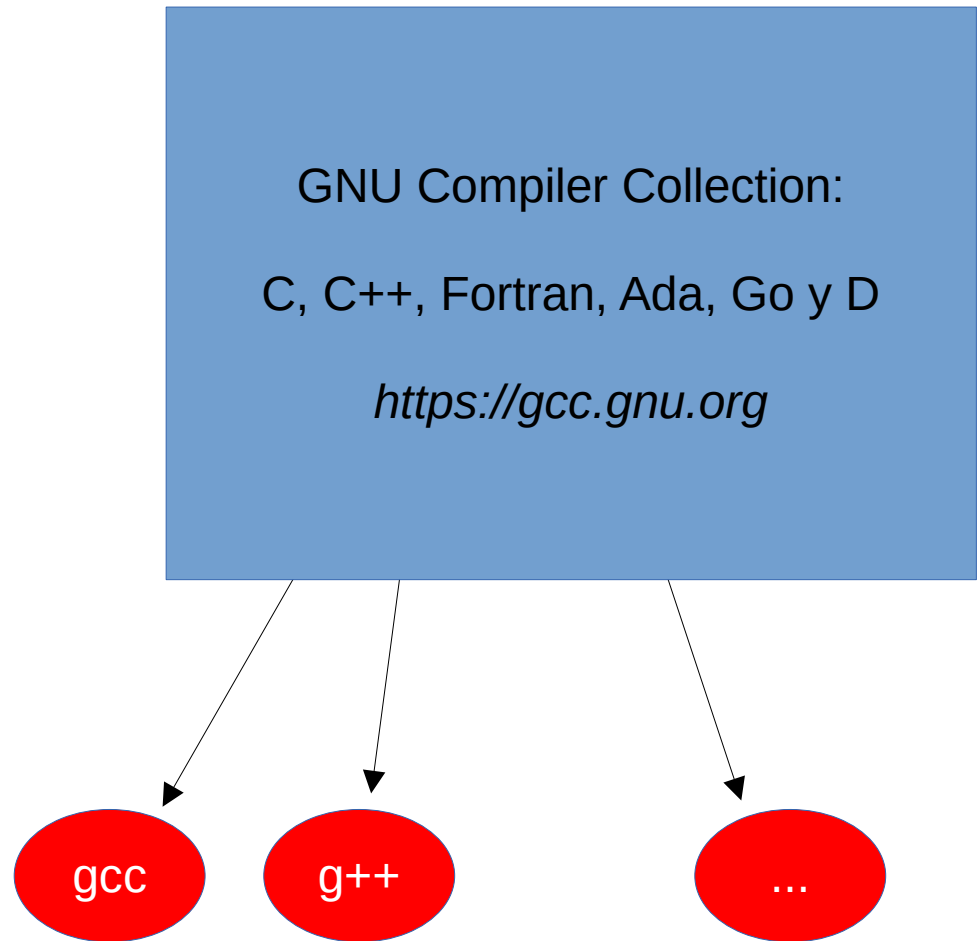
- <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#main>
- Etc.

Styleguide que usaremos en la asignatura - Resumen

1. Todos los identificadores deben ser descriptivos y en inglés. En general evitar doble underscore(_ _). Y nunca excesivamente largos.
2. **Ficheros** .h y .cc con nombres en minúsculas muy descriptivos y en inglés. Si es necesario usar varias palabras separadas preferentemente por underscores (_) aunque también se puede usar dashes(-). Evitar palabras clave de C++, tipos, clases y nombres de ficheros estándar.
3. Pensar bien lo que ponemos en los **comentarios** (deben ser en inglés). No comentar lo que se deduce claramente del código. Comentar lo que pasados unos días no entenderemos de la simple lectura del código.
4. Usar una buena indentación y márgenes del código fuente. Siempre tras if, for y while (un simple vistazo del código dice mucho sobre su autor/a).
5. The names of **variables** (including function parameters) and data members are all lowercase, with underscores between words. Data members of classes (but not structs) additionally have trailing underscores. For instance: a_local_variable, a_struct_data_member, a_class_data_member_. Los **espacios de nombres** también van en minúscula.
6. **Constantes** comienzan por 'k' y mixed case (const int kDaysInAWeek = 7)
7. **Funciones**: mixedcase. AddTableEntry(), DeleteUrl(), OpenFileOrDie(). También **clases** (MyMap, MyClass).
8. ALL_CAPS solo para **macros**.
9. Orden en las clases: public, protected, private (trailing underscore for private members)
10. Orden en las clases: constructores, destructores, funciones
11. Solo una instrucción por línea
12. Evitar 'goto', **includes** siempre al principio y solo los que se vayan a usar en ese fichero, usar siempre **guardas de inclusión**, etc.

C++ Compilers

1. GCC
2. clang
3. msvc
4. <https://isocpp.org/get-started>



El compilador GNU g++

Muestra el estándar por defecto

```
usuarioUCO@VTS2:~$ g++ -dM -E -x c++ /dev/null | grep -F __cplusplus  
  
#define __cplusplus 199711L  
  
(199711L es C++98)
```

Linux Debian/Mint/Ubuntu/etc.

```
$ g++ -dM -E -x c++ /dev/null | grep -F __cplusplus  
  
#define __cplusplus 201402L  
  
(201402L es gcc 10.2 y C++14)  
  
(201703L es gcc 11.4 y C++17)
```

Referencias:

- Definiciones de `__cplusplus`: <https://gist.github.com/ax3l/53db9fa8a4f4c21ecc5c4100c0d93c94>
- <https://stackoverflow.com/questions/44734397/which-c-standard-is-the-default-when-compiling-with-g>

El compilador GNU g++

g++ (GCC, *the GNU compiler collection*) <http://gcc.gnu.org>

```
$ g++ --version
```

```
g++ (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
```

```
Copyright (C) 2019 Free Software Foundation, Inc.
```

...

- Selección en la línea de comandos de g++ (-std=):
 - std=c++98 or -std=gnu++98 (también c++03, gnu++03)
 - std=c++11 or -std=gnu++11 (tb. de forma exp. en vers. anteriores de gcc: c++0x, gnu++0x)
 - std=c++14 or -std=gnu++14 (también c++1y, gnu++1y)
 - std=c++17 or -std=gnu++17 (c++1z, gnu++1z)
 - std=c++20 or -std=gnu++20 (c++2a, gnu++2a) desde mayo de 2022
 - std=c++23 or -std=gnu++23 (c++2b, gnu++2b)
- Estándares por defecto según la versión de gcc (UCO 10.2.0):
 - g++ 11 y 12 → c++17
 - g++ 6.1 al 10 → c++14
 - g++ prior to 6.1 → c++98

El compilador GNU g++ en la UCO

- Por defecto:

```
$ g++ --version
```

```
g++ (GCC) 5.3.0
```

```
Copyright (C) 2015 Free Software Foundation, Inc.
```

- Esta versión usa C++98 por defecto
- Para utilizar la versión 10 de g++, escribir en la terminal:

```
- $ .usegcc10-64
```

- O bien, para que se cargue siempre, añadir al final de .bashrc

```
export PATH=/bin64:/usr/local/bin64:/usr/local/opt/Qt-4.8.5/bin:/usr/local/bin:/bin:/usr/local/java/bin:/usr/local/eclipse:/usr/local/X11R7/bin:/usr/local/opt/gcc-10.2.0-32/bin:/usr/NX/bin:/usr/local/kde/bin:/usr/local/opt/libreoffice4.3/program
```

Salida de la compilación a fichero...

- Redireccionar la salida de errores de 'g++' o 'make' a un fichero para su análisis posterior:

```
$ g++ prueba.c > salida.txt 2>&1
```

o bien

```
$ make > salida.txt 2>&1
```

Más en moodle:

- <https://moodle.uco.es/m2425/mod/page/view.php?id=5709>

Comentarios

```
// one line comment
```

```
/* several
```

```
lines
```

```
comments
```

```
// including one line comment
```

```
And others lines
```

```
...
```

```
*/
```

Espacios de nombres - Namespaces

-----my-tools.h

```
namespace ns1 {  
    int a;  
    float b;  
    class c{...};  
}
```

Using something from inside ns1 namespace:

-----my-file.cpp

```
#include <my-tools.h>
```

```
int main() {
```

```
    ...
```

```
    ns1::a
```

```
    ns1::b
```

```
    ns1::c
```

```
    ...
```

```
}
```

RECOMMENDED!

To abbreviate it:

-----my-file.cpp

```
#include <my-tools.h>
```

```
using namespace ns1;
```

```
int main() {
```

```
    ...
```

```
    a
```

```
    b
```

```
    c
```

```
    ...
```

```
}
```



Espacios de nombres - Namespaces

Ventajas:

- 1) Evitan colisiones entre identificadores declarados en distintos ficheros/módulos
- 2) Evitan colisiones entre identificadores de distintos proyectos
- 3) Evitan colisiones entre identificadores de distintos desarrolladores
- 4) Mejora la organización y estructura del código en la programación modular

```
namespace ns1{
```

```
...
```

```
int v;
```

```
...
```

```
}
```

```
namespace ns2{
```

```
...
```

```
int v;
```

```
...
```

```
}
```

ns1::v y ns2::v no entran en conflicto, son diferentes variables

```
namespace ns3{
```

```
...
```

```
float v;
```

```
...
```

```
}
```

std::cin/std::cout

Objects *cin* and *cout* inside “std” namespace

```
#include <iostream>

int x;

std::cout<< "Write a number _";

std::cin >> x;
```

Abreviated:

```
#include <iostream>

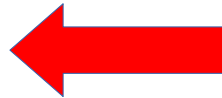
using namespace std;

int x;

cout<< "Write a number _";

cin >> x;

...
```



“std” es el espacio de nombres estándar de la librería estándar de C++ que se llama STL (Standard Template Library)

Fundamental C++ types

<code>void</code>	empty set of values			
<code>nullptr</code>	empty set of values for pointers (NULL is for int)			
<code>char</code>	signed/unsigned	Multibyte UTF-8 characters	<code>char8_t</code> , <code>char16_t</code> (UTF-16), <code>char32_t</code> (UTF-32)	
<code>int</code>	signed/unsigned	short (≥ 16), long (≥ 32)	long long (≥ 64)	<code>int8_t</code> , <code>int16_t</code> , <code>int32_t</code> , <code>int64_t</code> , <code>uint8_t</code> , <code>uint16_t</code> , <code>uint32_t</code> , <code>uint64_t</code>
<code>float</code>	IEEE-754 binary32 format	simple-precision		
<code>double</code>	IEEE-754 binary64 format	double-precision		
<code>long double</code>	IEEE-754 binary128 format	quadruple-precision		
<code>bool</code>	true/false			

- Lo tipos clásicos (int, float, etc.) dependen de la implementación
- Los tipos terminados en `_t` son tipos más precisos

Hay muchos tipos: siempre se debe ajustar bien.

La cadena de caracteres - std::string

- Cadenas UTF-8

```
#include <string>
...
std::string s1 = "hola";
std::string s2 = s1 + " mundo"; // concatenación
...
cout << s1[1]; // 'o'
cout << s1.length(); // 4
```

- Dispone de numerosos métodos:

- Asignar con =, concatenar con +
- Métodos length(), find()
- Obtener el char * con el método c_str() / data()
- <, >, <=, >=, ==, !=
- Etc.

std::string - conversión de tipos

- `std::stoi()`, `std::stol()` y `std::stoll()`, convierten un string a un int (4 bytes), long int (4 bytes) y long long int (al menos 8 bytes desde C++11).
- `std::stof()`, `std::stod()` y `std::stold()` convierten un string a un float (4 bytes), double (8 bytes) y long double (12 bytes desde C++11)
- `std::to_string()` permite convertir a `std::string` el dato pasado como parámetro sea int, float, etc.
- Más en:
 - <http://en.cppreference.com/w/>
 - <http://www.cplusplus.com/>

class

Implementación y ED internas
Atributos

```
class Point{
```

```
    private:
```

```
        int x_, y_;
```

```
    public:
```

```
        void Set(int x, int y) {x_=x; y_=y; }
```

```
        int Getx() {return x_;
```

```
        int Gety() {return y_;
```

```
};
```

ojo!

```
Point p;
```

```
p.Set(2, -1);
```

```
cout << p.Getx() << "\n";
```

```
cout << p.Gety() << endl;
```

Interfaz
Métodos

class

Implementación y ED internas
Atributos

```
class Point{
```

```
    private:
```

```
        int x_, y_;
```

```
    public:
```

```
        void Set(int x, int y) {x_=x; y_=y; }
```

```
        int Getx() {return x_; }
```

```
        int Gety() {return y_; }
```

Interfaz
Métodos

```
};
```

Creación del objeto/instancia

```
Point p;
```

```
p.Set(2, -1);
```

Llamada/invocación a un método
de la clase Point (se envía un mensaje al objeto 'p'
cuyo contenido es 2 y -1)

```
cout << p.Getx() << "\n";
```

```
cout << p.Gety() << endl;
```

class

Implementación y ED internas
Atributos

```
class Point{
```

```
    private:
```

```
        int x_, y_;
```

```
    public:
```

```
        void Set(int x, int y) {x_=x;y_=y;}
```

```
        int Getx() {return x_;
```

```
        int Gety() {return y_;
```

```
};
```

Interfaz
Métodos

```
Point p;
```

```
p.Set(2, -1);
```

modificador /setter

```
std::cout << p.Getx() << "\n";
```

```
std::cout << p.Gety() << endl;
```

```
std::cout << p.x_;
```

observador/getter

**ERROR acceso
denegado**

Constructor y destructor de una clase

```
// inicializa el objeto  
// puede tener parámetros  
Point::Point() {...}
```

```
// tareas de  
// finalización  
// del objeto
```

```
// puede tener  
parámetros
```

```
Point::~~Point() {...}
```

Tilde – ~



Tilde

Unicode: U+0007E

HTML entity: ˜ – HTML code: ~

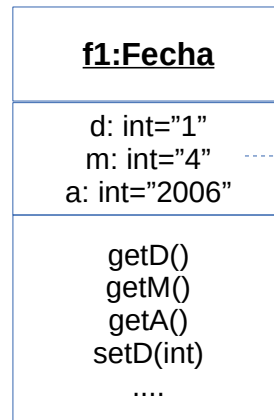
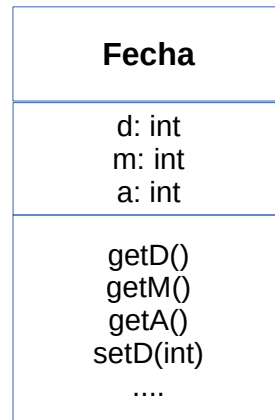
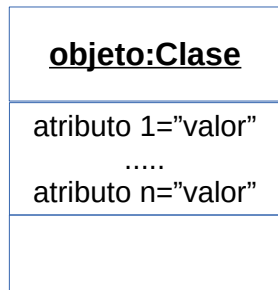
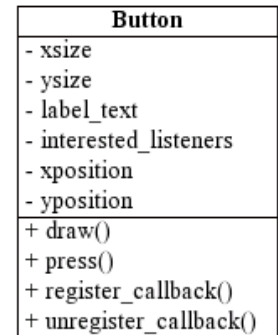
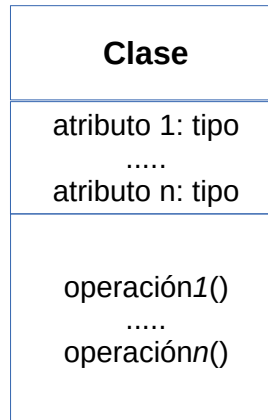
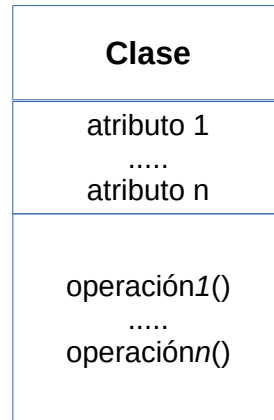
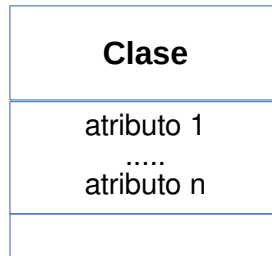
Also called the **swung dash**, **squiggly** or **twiddle**.

PC keystroke: ALT+0126

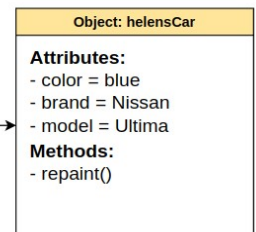
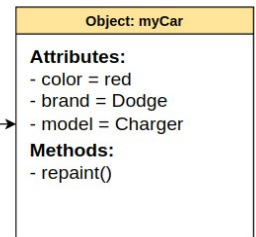
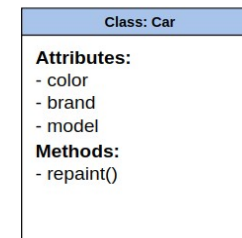
Used to indicate 'approximately' or 'around'. It can also indicate 'similar to'. In some languages, the tilde is placed over a letter to indicate a change in its pronunciation.

Notación gráfica con UML

clases y
objetos



anotación
(texto o gráfico)



include guards / guardas de inclusión

- Evitan redefiniciones al incluir ficheros de cabecera

```
#ifndef MY_TOOLS_H
```

```
#define MY_TOOLS_H
```

```
...
```

```
#endif
```

¡USAR SIEMPRE!

bool

- El tipo *bool*

valores: `true` y `false`

```
bool find(...) {
```

```
...
```

```
}
```

```
...
```

```
if (find()){
```

```
...
```

```
}
```

C language includes

```
#include <cstdio> // Biblioteca estandar I/O  
                // de C (ya no es stdio.h)
```

Lo mismo con:

`cmath`

`cstdlib`

`...`

typedef / struct

- En C++ no es necesario el uso de `typedef` (una declaración `struct` permite declarar datos de ese tipo)

```
...  
struct Driver  
{  
    char *name[50];  
    int age;  
};  
...  
Driver p;
```

cout

```
#include <iostream>
#include <string>
std::string name= "Juan";
std::cout << "hola" << name << "\n";
```

Clase string en C++
(cadenas de caracteres)

Lo mismo que:
std::endl

const

- Declaración de constantes con “const” (necesaria inicialización):

```
...  
const int kSpeedOfLight=300000;
```

```
...  
const float kPI=3.14159;
```

```
...
```

(por convención, se suelen identificar con una ‘k’ al principio de su nombre)

Funciones const/getters/observadores

```
class Point{
    private:
        int x_, y_;
    public:
        Point(int x, int y){x_=x;y_=y;}
        void Set(int x, int y){x_=x;y_=y;}
        int Getx() const {return x_;}
        int Gety() const {return y_;}
};

Point p(0,0);
p.Set(2,-1);
std::cout << p.Getx() << "\n";
std::cout << p.Gety() << endl;
```

Los observadores/*getters*.
No modifican el objeto.

observador/getter



Funciones const

```
int Date::getDay() const {return day_};
```

```
int Date::setDay(int d);
```

si se usa .h y .cc hay que ponerlo en los dos

```
const Date birthDate(1,1,1970);
```

```
birthDate.getDay();
```

```
birthDate.setDay(5); //setDay() is not const
```

Parámetros const

IMPORTANTE: si una función recibe un objeto const, como parámetro solo podrá invocar métodos const de ese objeto

```
Date d(23, 9, 2024);  
int funcion(const Date p) {  
    birthDate.getDay();  
    birthDate.setDay(5); //ERROR setDay() is not const  
}
```

Al compilar el programa con g++ nos indicará el error

Parámetros const

```
int f(const int *v)
{
    ...
    v[2]=55; // error: asignación a ubicación de sólo lectura
    ...
}
```

Al compilar el programa con g++ nos indicará el error

Parámetros por defecto

Siempre al final de la lista de parámetros y solo en la declaración de la función (solo en el fichero .h)

```
void load(float peso, float volumen, int tipo=0, int subtipo=2);// Ok
void unload(float peso, int tipo=0, float volumen, int subtipo=2);// Mal,
// ya que hay un parámetro sin valor por
// defecto ("volumen") detrás de uno
// con valor por defecto.
```

- load() está bien definida.
- unload() no está bien definida.

Funciones inline

- Funciones *inline* (en línea) optimizan la ejecución de funciones breves:
 - El compilador sustituye su llamada con el código de la función ahorrando el tiempo empleado en la llamada, cambio de contexto, etc.
- Normalmente debe hacerse siempre en la declaración de la clase (en el fichero .h). La palabra clave “inline” debe escribirse delante de la definición de la función.
- Si simplemente se escribe el código en el .h, también se considera inline.
- Si la función no es breve, C++ descartará automáticamente la solicitud de función inline.

Funciones inline

- Es recomendable realizar la declaración en .h

Declaración 1

```
// Fichero dados.h
#ifndef DADOS_H
#define DADOS_H
class Datos{
    private:
        . . .
    public:
        inline int getDado1() { return d1_; }
};
#endif
```

Declaración 2:

```
// Fichero dados.h
#ifndef DADOS_H
#define DADOS_H
class Datos{
    private:
        . . .
    public:
        int getDado1();
};

inline int Datos::getDado1() {
    return d1_;
}
#endif
```

Funciones inline

- Es recomendable realizar la declaración en .h

Declaración 3

```
// Fichero dados.h
#ifndef DADOS_H
#define DADOS_H
class Datos{
    private:
        . . .
    public:
        inline int getDado1();
};
#endif
```

```
// Fichero dados.cc
```

```
inline int Datos::getDado1() {
    return d1_;
}
```


Funciones inline

- También pueden ser inline funciones independientes, es decir, funciones que no pertenecen a ninguna clase.

Referencias

- Referencias (son como un ‘alias’):

`int var;`



A diagram showing a memory cell. Above the cell is the label 'var'. The cell itself is a blue rectangle containing the text '--'.

`int &p=var;`



A diagram showing a memory cell. Above the cell is the label 'var'. To the left of the cell is the label 'p'. The cell itself is a blue rectangle containing the text '--'.

`p= 77;`



A diagram showing a memory cell. Above the cell is the label 'var'. To the left of the cell is the label 'p'. The cell itself is a blue rectangle containing the text '77'.

- Con “p” se puede operar exactamente igual que con “var”

Referencias

- C++ hereda los punteros de C por compatibilidad.
- Diferencias entre referencias y punteros:
 - Las referencias deben asignarse a una zona de memoria legítima. Está asegurado que siempre apuntan a algo válido (por seguridad).
 - No pueden reasignarse a otra variable o zona de memoria (podría asignarse a algo inválido).
 - Simplifican a la vez que mantienen la eficiencia de los punteros (sobre todo en el paso de parámetros).
 - no son punteros.

Referencias

- Las referencias se utilizan mucho en el paso de parámetros.
- Veamos un ejercicio:
 - Hacer una función que intercambie el valor de 2 enteros "a" y "b".

Referencias

- En “C”:

```
void intercambia(int *a, int *b){  
    int aux=*a;  
    *a=*b;  
    *b=aux;  
}
```

Diagram illustrating the excessive use of pointers and addresses in the provided C code. Red arrows point from the central text to the following elements:

- The pointer parameter `*a` in the function signature.
- The pointer parameter `*b` in the function signature.
- The dereferenced pointer `*a` in the assignment `*a=*b`.
- The dereferenced pointer `*b` in the assignment `*b=aux`.
- The address-of operator `&` in the function call `&x`.
- The address-of operator `&` in the function call `&y`.

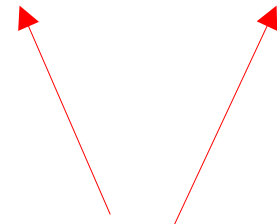
*hay un uso
excesivo de * y &*

Y la llamada: `intercambia(&x,&y)`

Referencias

- Ahora usando “referencias” en C++:

```
void intercambia(int &a, int &b){  
    int aux=a;  
    a=b;  
    b=aux;  
}
```



*único uso de &
(mucho más simple
que con punteros)*

Y la llamada: `intercambia(x,y)`

Paso de objetos usando referencias

- Usar parámetros por referencia tiene **varias ventajas**:
 - No se realiza copia del objeto por lo que se ahorra memoria
 - Al no realizarse la copia se ahorra tiempo de cómputo
 - No es necesario el uso de punteros por lo que se simplifica el código

```
void funcion(Person &q){ . . . }
```

- Si el parámetro no se modifica en la función usaremos:
 - **referencias constantes**:

```
void funcion(const Person &q){ . . . }
```

arrays

- Arrays clásicas/estáticas estilo C. Son rápidas y eficientes y también se pueden usar en C++
 - una dimensión []
 - dos dimensiones [][]
 - tres dimensiones [][][]
 - etc.
 - realloc() mueve a otra localización mayor y conserva valores, etc.
- Si queremos más funcionalidad y/o que sea memoria dinámica:
 - std::array (estáticas muy parecidas a las de estilo C, pero con la semántica y funcionalidades añadidas de la STL)
 - std::vector (dinámicas y más lentos pero admite inserciones y adaptaciones de tamaño pudiendo ser más cómodos y rápidos en estos casos que std::array)

Un array estático más funcional - `std::array`

- Más eficiente que `std::vector` para arrays de tamaño fijo (Se almacena en la pila/stack).
- Los elementos se almacenan contiguos
- Uni o multidimensional manteniendo el acceso tipo C: `[]`, `[][]`, `[][][]`, etc.
- También se permite acceso con punteros, iteradores, `range-for`, etc.
- Array de 3 enteros:
 - `std::array<int, 3> a = {1,2,3};`
- Array bidimensional de 3x3 enteros:
 - `std::array<std::array<int, 3>, 3> arr = {{{5, 8, 2}, {8, 3, 1}, {5, 3, 9}}};`

std::array (constant size/static)

```
// CPP program to demonstrate working of array
#include <algorithm>
#include <array>
#include <iostream>
#include <iterator>
#include <string>
using namespace std;

int main() {

    // construction uses aggregate initialization
    // double-braces required
    array<int, 5> ar1{{3, 4, 5, 1, 2}};
    array<int, 5> ar2 = {1, 2, 3, 4, 5};
    array<string, 2> ar3 = {{string("a"), "b"}};

    cout << "Sizes of arrays are" << endl;
    cout << ar1.size() << endl;
    cout << ar2.size() << endl;
    cout << ar3.size() << endl;

    cout << "\nInitial ar1 : ";
    for (auto i : ar1)
        cout << i << ' ';

    // container operations are supported
    sort(ar1.begin(), ar1.end());

    cout << "\nsorted ar1 : ";
    for (auto i : ar1)
        cout << i << ' ';

    // Filling ar2 with 10
    ar2.fill(10);

    cout << "\nFilled ar2 : ";
    for (auto i : ar2)
        cout << i << ' ';

    // ranged for loop is supported
    cout << "\nar3 : ";
    for (auto &s : ar3)
        cout << s << ' ';

    return 0;
}
```

Output:
Sizes of arrays are
5
5
2
Initial ar1 : 3 4 5 1 2
sorted ar1 : 1 2 3 4 5
Filled ar2 : 10 10 10 10 10
ar3 : a b

Fuente: <https://www.geeksforgeeks.org/stdarray-in-cpp/>

Un array dinámico y funcional - `std::vector`

- Los elementos también se almacenan contiguos pero el tamaño es dinámico y se adapta en tiempo de ejecución bajo demanda.
- Menos eficiente que `std::array` para arrays de tamaño fijo (se almacena en heap).
- También se permite acceso con punteros, iteradores, range-for, etc.
- También uni o multidimensional.
- Multitud de funciones: `insert()`, `erase()`, `replace()`, `size()`, `clear()`, `swap()`, `empty()`, `push_back()`, `pop_back()`, `front()`, `back()`, etc.

[plantilla/template, operate with generic types]

```
#include <iostream>
#include <vector>
std::vector<int> v = { 7, 5, 16, 8 };
v.push_back(25);
v.push_back(13);
std::cout << v[0] << ", " << v[1] << v[2] << std::endl;
v.clear(); // Borra todos los elementos
```


Un array dinámico y funcional - `std::vector`

- Al añadir un nuevo elemento:
 - Se adapta el tamaño del vector si fuera necesario
 - Se crea un nuevo objeto del tipo base donde se hace una copia el elemento a añadir.
- Al borrar un elemento:
 - Se elimina de la lista liberando su espacio.

std::vector y range-for

```
for (int n : v) {  
    std::cout << n << ", ";  
}  
std::cout << "\n";
```

[Con “n” se hace un copia en cada iteración. La referencia “&n” es más eficiente.]



Formas de crear std::vector

```
// Empty vector
```

```
vector<int> vect;
```

```
vect.push_back(10);
```

```
vect.push_back(20);
```

```
vect.push_back(30);
```

```
// Vector of size n with all values as 10.
```

```
vector<int> vect(n, 10);
```

```
vector<int> vect1{ 10, 20, 30 };
```

```
// Initializaing from another vector
```

```
vector<int> vect2(vect1.begin(), vect1.end());
```

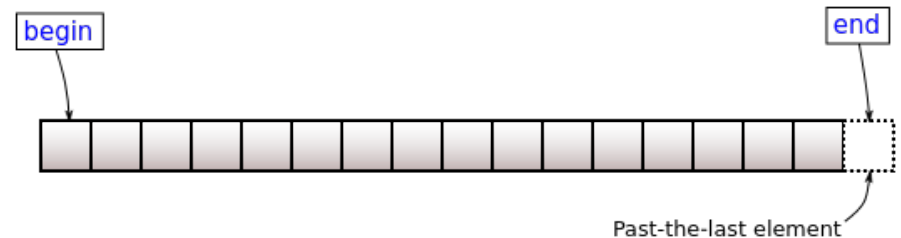
`std::vector< ... >iterator it`

- `*it`, `v.begin()`, `v.end()` y aritmética de punteros

[plantilla/template,
operate with generic
types]

[El iterador puede llamarse
“it” o tener cualquier otro
nombre de identificador]

```
#include <iostream>
#include <vector>
std::vector<int> v = { 7, 5, 16, 8 };
std::vector<int>::iterator it = v.begin();
std::cout << *it << std::endl;
it++;
std::cout << *it << std::endl;
```



Declaración de variables y **auto**

- En cualquier lugar (siempre antes de su uso)

- **auto**

- `auto a = 'a';`
- `auto i = 123;`
- `auto b = true;`
- `auto cad = "hola"`
- `auto cad2 = std::string("hola");`

`char, int, bool, char *, std::string`

```
#include <typeinfo>
```

```
std::cout << typeid(cad).name() << std::endl;
```


auto y range-for

```
std::vector<std::string> my_vector;  
my_vector.push_back("foo");  
my_vector.push_back("bar");  
my_vector.push_back("baz");
```

```
for (int i=0;i<3;i++)  
{  
    cout << my_vector[i]<<std::endl;  
}
```

auto y range-for

```
std::vector<std::string> my_vector;
```

```
my_vector.push_back("foo");
```

```
my_vector.push_back("bar");
```

```
my_vector.push_back("baz");
```

```
std::vector<std::string>::iterator it;
```

```
for (it=my_vector.begin(); it!=my_vector.end(); it++)
```

```
{
```

```
    cout << *it <<std::endl;
```

```
}
```

auto y range-for

```
std::vector<std::string> my_vector;  
my_vector.push_back("foo");  
my_vector.push_back("bar");  
my_vector.push_back("baz");
```

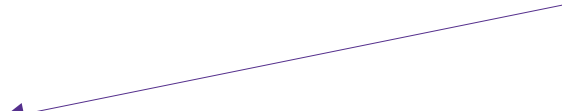
```
for (auto it=my_vector.begin(); it!=my_vector.end(); it++)  
{  
    cout << *it <<std::endl;  
}
```

auto y range-for

```
std::vector<std::string> my_vector;  
my_vector.push_back("foo");  
my_vector.push_back("bar");  
my_vector.push_back("baz");
```

[Una copia]

```
for (std::string e: my_vector)  
{  
    cout << e <<std::endl;  
}
```

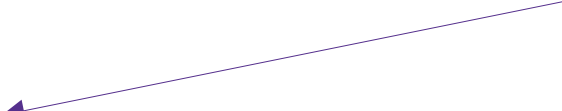


auto y range-for

```
std::vector<std::string> my_vector;  
my_vector.push_back("foo");  
my_vector.push_back("bar");  
my_vector.push_back("baz");
```

[Una referencia]

```
for (std::string &e: my_vector)  
{  
    e = e + " modificada";  
    cout << e << std::endl;  
}
```

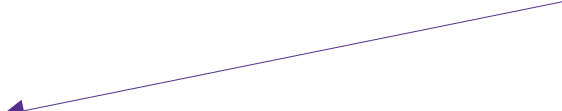


auto y range-for

```
std::vector<std::string> my_vector;  
my_vector.push_back("foo");  
my_vector.push_back("bar");  
my_vector.push_back("baz");
```

[Una referencia]

```
for (auto &e: my_vector)  
{  
    e = e + " modificada";  
    cout << e << std::endl;  
}
```



STL: Standard Template Library

Contiene: funciones, algoritmos, iteradores y contenedores:

- Vector
 - <http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html#VECTOR>
- List (enlaces en la web de la asignatura):
 - <http://www.cplusplus.com/reference/list/list/>
 - <http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html#LIST>
- stack
- queue
- set, map, hash, etc
- Más en:
 - <http://en.cppreference.com/w/>
 - <http://www.cplusplus.com/>

Una lista dinámica – `std::list` - 1/3

- Eficiente en random insert/remove (`std::vector` debe desplazar el resto de elementos...)
- `std::vector` más rápida de recorrer, encontrar y ordenar
- `std::array` aún más rápida y `[]` aún más, pero cada una tiene sus características.
- Más en:
 - <http://en.cppreference.com/w/>
 - <http://www.cplusplus.com/>

Una lista dinámica – `std::list` - 2/3

- Al añadir un nuevo elemento:
 - Se crea un nuevo objeto del tipo base donde se hace una copia el elemento a añadir.
- Al borrar un elemento:
 - Se elimina de la lista liberando su espacio.

Una lista dinámica – `std::list` - 3/3

- ¡Ojo!: `std::list` tiene su propio funcionamiento
- Por ejemplo: "Removing Elements From a List while Iterating through it":
 - <https://thispointer.com/how-to-remove-elements-from-a-list-while-iterating/>

enum class

- In 'C ', two enumerations cannot share the same names:

```
enum month { jan=1, may=5, jun, jul };  
enum color { rojo, verde, azul, rosa};  
enum flor { margarita, rosa, lila};
```

// 'rosa' conflicts with a previous declaration

- No variable can have a name which is already in some enumeration.
- Enums are not type-safe (no tienen su propio tipo, son convertidos a int)

enum class

C++11 has introduced **enum classes**. Class enum doesn't allow implicit conversion to int, and also doesn't compare enumerators from different enumerations.

```
// C++ program to demonstrate working
// of Enum Classes
#include <iostream>
using namespace std;
```

```
int main()
{
    enum class Color { Red,
                      Green,
                      Blue };

    enum class Color2 { Red,
                       Black,
                       White };

    enum class People { Good,
                       Bad };

    // An enum value can now be used
    // to create variables
    int Green = 10;
```

```
// Instantiating the Enum Class
```

```
Color x = Color::Green;
```

```
// Comparison now is completely type-
```

safe

```
if (x == Color::Red)
    cout << "It's Red\n";
else
    cout << "It's not Red\n";
```

```
People p = People::Good;
```

```
if (p == People::Bad)
    cout << "Bad people\n";
else
    cout << "Good people\n";
```

```
// error por diferentes tipos
```

```
// if(x == p)
```

```
// cout<<"red is equal to good";
```

```
// won't work as there is no
```

```
// implicit conversion to int
```

```
// cout<< x;
```

```
cout << int(x); // sale 0
```

```
return 0;
```

```
}
```

Sobrecarga de funciones

- Distintas funciones con el mismo nombre
- Muy útil y cómodo pues mejora las interfaces

```
void swap(int &a, int &b);  
void swap(float &a, float &b);
```

- También dentro de una misma clase

```
class Grupo{  
    public:  
        bool DeleteUser(User u);  
        bool DeleteUser(std::string DNI);  
    . . .  
};
```

Sobrecarga de funciones

- También hay sobrecarga de constructores:

```
class Car{  
    public:  
        Car(std::string name);  
        Car(int id);
```

```
    . . .
```

```
};
```

Iniciadores (de miembros)

```
class A{  
private:  
    int x_, y_;  
    ...  
public:  
    A() : x_(1), y_(1) {};  
    ...  
};
```

(si la declaración de la función está en el .h y el cuerpo en el .cc, debe ir en .cc)

Iniciadores (de miembros)

```
class A{
private:
    int n_;
public:
    A(int x) {n_=x;}
};

class B{
private:
    int x_, y_;
    A obj_;
    ...
public:
    B() : x_(1), y_(1),
    obj_(10) {};
    ...
};
```

parám. obligatorio

O bien mediante un parámetro:

```
class B{
private:
    int x_, y_;
    A obj_;
    ...
public:
    B(int x) : x_(1), y_(1),
    obj_(x) {};
    ...
};
```

(si la declaración de la función está en el .h y el cuerpo en el .cc, debe ir en .cc)

Constructor de copia

obj2
a=
b=

obj1
a=7
b=3

Copia por defecto de un objeto:

```
MiClase obj1;
```

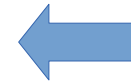
```
MiClase obj2(obj1);
```

```
MiClase obj2=obj1;
```

```
...return *obj1
```

Iguals (se ejecuta el constructor de copia)

obj2
a=7
b=3



obj1
a=7
b=3

constructor de copia por defecto: *solo asignaciones*

Constructor de copia propio:

```
MiClase(const MiClase &e) { ... }
```

(necesario cuando solo las asignaciones no es suficiente)

(habrá que hacer parecido con el operador de asignación por defecto, =)

Constructor de copia

Si queremos que el comportamiento sea el comportamiento por defecto es conveniente declararlo explícitamente:

```
MiClase(const MiClase&)=default;
```

(en versiones nuevas del compilador, si no se hace esto nos da error/warning)

C++ Compound Types

Not fundamental types (int, float, etc.)

Defined in terms of another C++ type (array, struct, etc.).

Some new compound types:

- pair
- tuple
- map

pair

is a class template that provides a way to store two heterogeneous objects ('first' & 'second') as a single unit.

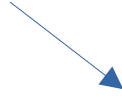
```
pair <int, string> p1;  
p1.first=200;  
p1.second="hola";  
cout << p1.first << endl;  
cout << p1.second << endl;
```

O bien:

```
auto p3 = std::make_pair(200, string("hola"));
```

tuple

'mytuple()' es una función
que devuelve una tupla



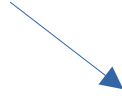
- A fixed-size collection of heterogeneous values. It is a generalization of `std::pair`.

```
std::tuple<char, int, bool> t;  
t=make_tuple('a', 10, true);  
cout << std::get<0>(t) << ", " <<  
      std::get<1>(t) << ", " <<  
      std::get<2>(t) << "\n";
```

- Se puede acceder a cada elemento, saber su tamaño, swap, concatenar tuplas, etc. (más en manual de `std::tuple`)

tuple

'mytuple()' es una función
que devuelve una tupla

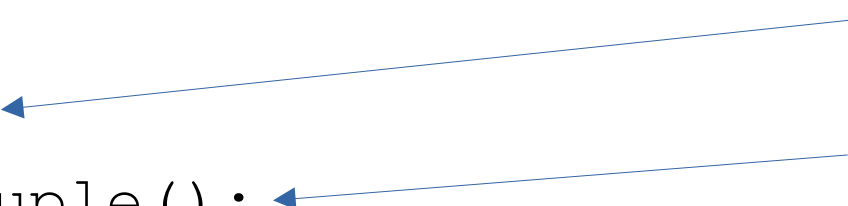


A fixed-size collection of heterogeneous values. It is a generalization of `std::pair`.

```
std::tuple<char, int, bool> mytuple() {  
    char a = 'a';  
    int i = 123;  
    bool b = true;  
    return std::make_tuple(a, i, b);  
}
```

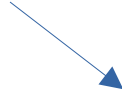
```
auto t = mytuple();  
auto [x, y, z] = mytuple();
```

't' es una tupla
'x' es un char
'y' es un int
'z' es un bool



tuple

'mytuple()' es una función
que devuelve una tupla



```
std::tuple<double, char, std::string> get_student(int id)
{
    switch (id)
    {
        case 0: return {3.8, 'A', "Lisa Simpson"};
        case 1: return {2.9, 'C', "Milhouse Van Houten"};
        case 2: return {1.7, 'D', "Ralph Wiggum"};
        case 3: return {0.6, 'F', "Bart Simpson"};
    }

    throw std::invalid_argument("id");
}
```

Al devolver una tupla, el error lo podemos
lanzar como una excepción.

```
...
try{
    const auto [gpa4, grade4, name4] = get_student(4);
}
catch (std::invalid_argument const& ex)
{
    std::cout << "#1: " << ex.what() << '\n';
}
```

map

```
std::map<char,int> mymap;
```

```
mymap['a']=10;
```

```
mymap['b']=20;
```

```
mymap['c']=30;
```

```
mymap['d']=40;
```

```
map<char, int>::iterator it;
```

```
for(it=mymap.begin(); it!=mymap.end(); ++it){  
    cout << it->first << " => " << it->second << '\n';  
}
```

Insert:

```
std::map<char,int> mymap;
```

```
auto [itelem, success] = mymap.insert(std::pair('a', 100)); // returns a std::pair
```

```
if (success) {  
    std::cout << "success!" << std::endl;  
}
```

```
else {  
    std::cout << "error!" << std::endl;  
}
```


map

```
std::map<char,int> mymap;
```

```
mymap['a']=10;
```

```
mymap['b']=20;
```

```
mymap['c']=30;
```

```
mymap['d']=40;
```

```
map<char, int>::iterator it;
```

```
for(it=mymap.begin(); it!=mymap.end(); ++it){  
    cout << it->first << " => " << it->second << '\n';  
}
```

Insert:

```
std::map<char,int> mymap;
```

```
auto [itelem, success] = mymap.insert(std::pair('a', 100)); // returns a std::pair
```

```
if (success) {  
    std::cout << "success!" << std::endl;  
}
```

```
else {  
    std::cout << "error!" << std::endl;  
}
```

Herencia

- Uno de los conceptos más importante de la POO.
- Básicamente consiste en definir una clase a partir de otra previamente definida.
- La herencia explota la reutilización, que es una gran potencia en el diseño de software.
- Relación jerárquica entre entidades del sistema.

Herencia

```
class Vehicle{
public:
    void SetWheels(int w);
    int GetWheels();
    void SetPassenger(int p);
    int GetPassenger();
private:
    int wheels_;
    int passengers_;
};
```

vehicle.h

```
#include "vehicle.h"
class Truck : public Vehicle{
public:
    void SetLoad(float l);
    float GetLoad();
private:
    float load_;
};
```

truck.h

```
#include "vehicle.h"
class Motorcycle : public Vehicle{
public:
    void SetSidecar(bool s);
    bool GetSidecar();
private:
    bool sidecar_;
};
```

motorcycle.h

```
#include "truck.h"
#include "motorcycle.h"

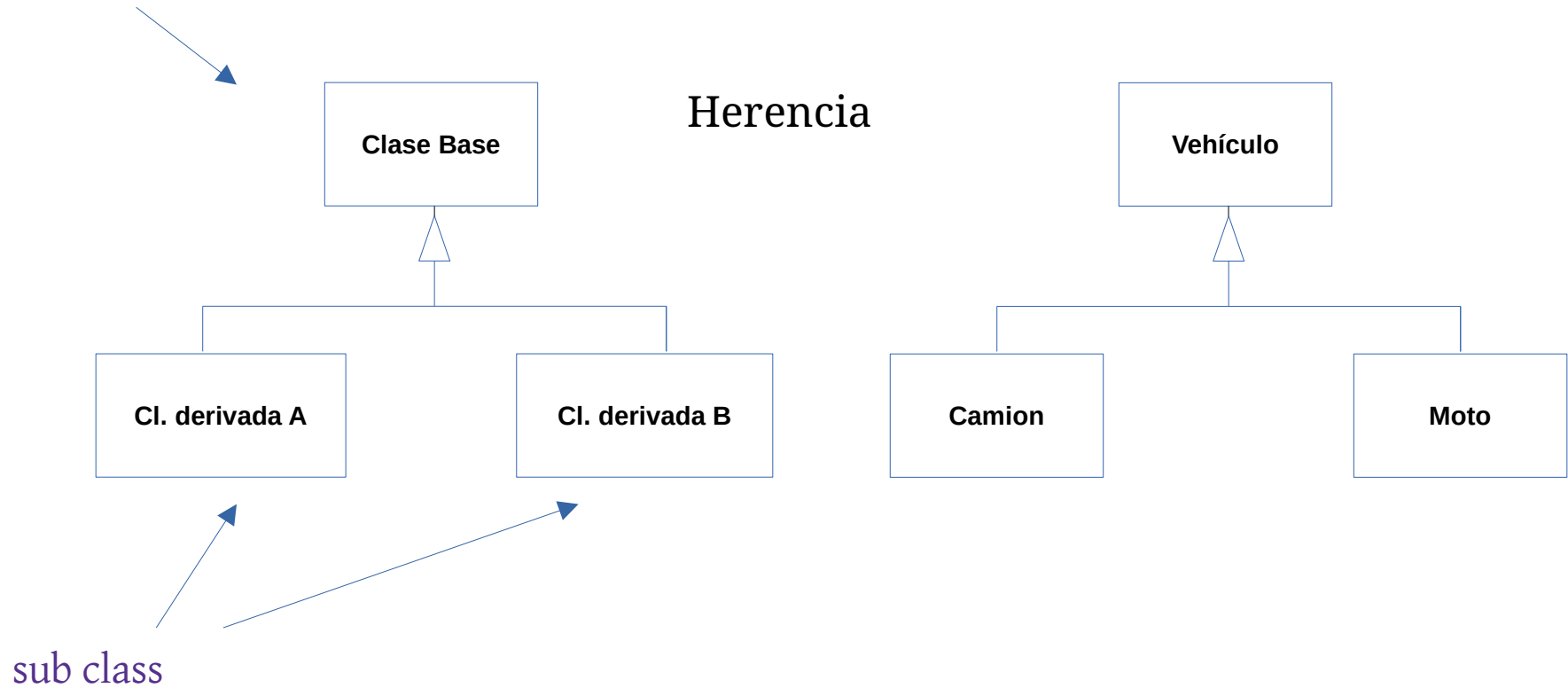
int main()
{
    Truck c;
    Motorcycle m;
    ...
    cout << "ruedas del camión = " << c.GetWheels();
    cout << "ruedas de la moto = " << m.GetWheels();
    ...
    cout << "carga del camión = " << c.GetLoad();
    ...
    cout << "tiene sidecar? " << m.GetSidecar();
    ...
}
```

plant-main.cc

Notación gráfica en UML

super class

Herencia



Herencia, iniciadores base

```
class A{
private:
    int n_;
public:
    A(int x){n_=x;}
}
class B: public A{
...};
```

Iniciadores base: si el constructor de la clase A tiene un parámetro obligatorio, se debe enviar desde la clase derivada usando un iniciador de la clase base:

```
B::B(int y, . . .): A(y){...}; // se envía un parám. de B::B
O bien
B::B(): A(10){...}; // se envía un literal
```

Herencia, iniciadores base

```
class A{
private:
    int n_;
public:
    A(int x){n_=x;}
}
class B: public A{
...};
```

Si en el constructor 'x' tuviese un valor por defecto:

```
A(int x=0){n_=x;}
```

Se podrían declarar objetos sin pasar 'x':

```
A obj1;
```

Entonces sería opcional el iniciador en B:

```
B::B(...) {...}
```

Si el constructor de la clase base tiene un parámetro obligatorio, éste debe pasarse en un iniciador base de la clase derivada.

Iniciadores base: ¿dónde se declaran?

(si la declaración del constructor está en el .h y el cuerpo en el .cc, debe ir en .cc)

Variables miembro static

Static members: only one copy of the static member is shared by all objects of a class in a program

```
class A{  
    private:  
        static int var;  
    ...  
};  
...  
int A::var=4;    // se usa como A::var  
...
```


Constantes miembro static

Son las únicas que pueden inicializarse en la declaración.

```
class A{  
    private:  
        const static int kval=2336547;  
    ...  
};
```

Funciones miembro static

Static Function Members. By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator `::` before the name of the function.

Invocar una función estática:

```
nombre_clase::nombre_funcion()
```

Uso:

- Agrupar funciones independientes en un mismo módulo
- Para poder usarlas sin tener que declarar objetos de su clase

Inicialización de miembros (y de miembros constantes)

```
Class A{
```

```
private:
```

```
    const int i_; // necesita inicializarse
```

```
    B obj1_; //necesita param.
```

```
    C obj2_; // necesita param.
```

```
public:
```

```
    A(int edad, float peso):i_(7),  
        obj1_(peso), obj2_(edad){...}
```

```
    ...
```

```
};
```

El puntero this

```
class A{  
    private:  
        int i_;  
    ...  
};
```

Accediendo al dato desde un método de la clase:

```
i_ = x;  
es igual que  
this->i_ = x; //puntero al objeto actual (this)
```

Todos los métodos de una clase reciben el puntero this

Sobrecarga de operadores

```
class Punto{
public:
    Punto(int x, int y){x_=x; y_=y;};
    int getX(){return x_};
    int getY(){return y_};
    Punto operator=(const Punto &p);
    Punto operator+(const Punto &p);
    Punto operator++(void); // para ++p
    Punto operator++(int); // para p++
private:
    int x_, y_;
};

Punto Punto::operator=(const Punto &p)
{
    x_ = p.x_;
    y_ = p.y_;
    return *this;
}

Punto Punto::operator+(const Punto &p)
{
    Punto aux;
    aux.x_ = x_ + p.x_;
    aux.y_ = y_ + p.y_;
    return aux;
}
```

```
Punto Punto::operator++(void)// ++b;
{
    x_++;
    y_++;
    return *this;
}

Punto Punto::operator++(int)// b++;
{
    Punto aux=*this; // OJO!!
    x_++;
    y_++;
    return aux;
}

.....
int main(void)
{
    Punto a(1,2),b(10,10),c(0,0), d(3,3);
    d=c+a+b;
    c=a+b+c;
    b++;
    c.out << "b.x= " << b.getX() << "b.y= "
    << b.getY() << endl;
    c.out << "c.x= " << c.getX() << "c.y= "
    << c.getY() << endl;
}
```

Sobrecarga de operadores y constructores de copia

Cuando sobrecargamos el operador = es conveniente hacer un constructor de copia por defecto. O bien:

```
Punto(const Punto& o) {x_=o.coordx_; y_=o.coordy_};
```

O lo que sería lo mismo y más breve:

```
Punto(const Punto&)=default;
```

De otra forma nos daría error/warning en versiones nuevas de los compiladores.

Esto es debido a que si sobrecargamos el operador de asignación seguramente es necesario que sobrecarguemos el constructor de copia y el destructor y son frecuentes los errores. Esta regla se llama “Rule of Three”:

- “If a class requires a user-defined destructor, a user-defined copy constructor, or a user-defined copy assignment operator, it almost certainly requires all three.”
(https://en.cppreference.com/w/cpp/language/rule_of_three)

Funciones friend

```
class Circulo;

class Linea{
private:
    int color_;
public:
    friend MismoColor(const Linea &l,
                      const Circulo &c);
    ...
};

class Circulo{
private:
    int color_;
public:
    friend MismoColor(const Linea &l,
                      const Circulo c);
    ...
};
```

```
int MismoColor(const Linea &l,
               const Circulo &c)
{
    if (l.color_==c.color_)
        return 1;
    else
        return 0;
}
```

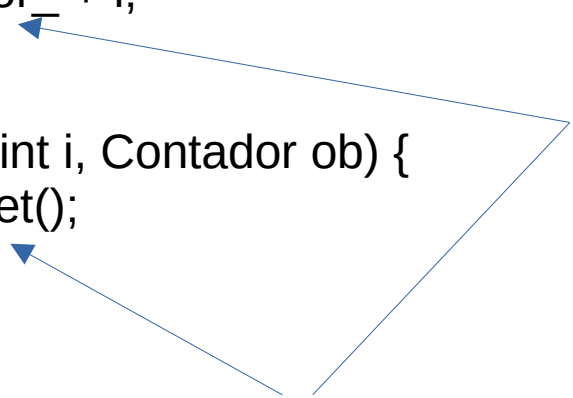
Las funciones friend no reciben el puntero `this`

Sobrecarga de operadores con funciones friend

```
class Contador{  
private:  
    int valor_;  
public:  
    Contador(){valor_=0;}  
    int Contador::operator+(int x)  
        {return valor_+x;}  
...  
};  
  
Contador c;  
int i;  
...  
...  
i = c + 10;  
i = 10 + c // No funciona
```

SOLUCIÓN:

```
class Contador{  
public:  
    friend int operator+(Contador ob, int i);  
    friend int operator+(int i, Contador ob);  
.....  
private:  
    int valor_;  
};  
  
// esta podría no ser friend  
int operator+(Contador ob, int i) {  
    return ob.valor_ + i;  
}  
  
int operator+(int i, Contador ob) {  
    return i+ob.get();  
}
```



Accederá o no a la parte privada, pero debe ser friend por incorporar el tipo de ambos operandos.

Herencia private

```
struct Node{
    int info;
    Node *next;
};

class List{
private:
    Node *head_, *tail_, *cursor_;
    int t_;
public:
    List(){head_=tail_=cursor_=NULL;t_=0;};
    ~List();
    int Size(){return t_};
    bool Empty(){return t_?false:true;};
    void Reset(){cursor_=head_};
    int Next();
    int Get(int &info);
    int Set(int valor);
    int Insert(int x);
    int InsertBegin(int x);
    int Remove();
};
```

Herencia private

```
class Stack: private List{
public:
    Stack();
    int Push(int i);
    int Pop();
    bool Empty();
    int Top(int &info);
    ~Stack();
private:
};
```

```
Stack::Stack()
{
}
```

```
Stack::~~Stack()
{
}
```

```
int Stack::Push(int i)
{
    return InsertBegin(i);
}
```

```
int Stack::Pop()
{
    int i;
    Reset();
    if (get(i) != ERROR)
    {
        Remove();
        return i;
    }
    else return ERROR;
}
```

```
bool Stack::Empty()
{
    return List::Empty();
}
```

```
int Stack::Top(int &info)
{
    reset();
    if (List::Empty()) return 0;
    else Get(info);
    return 1;
}
```

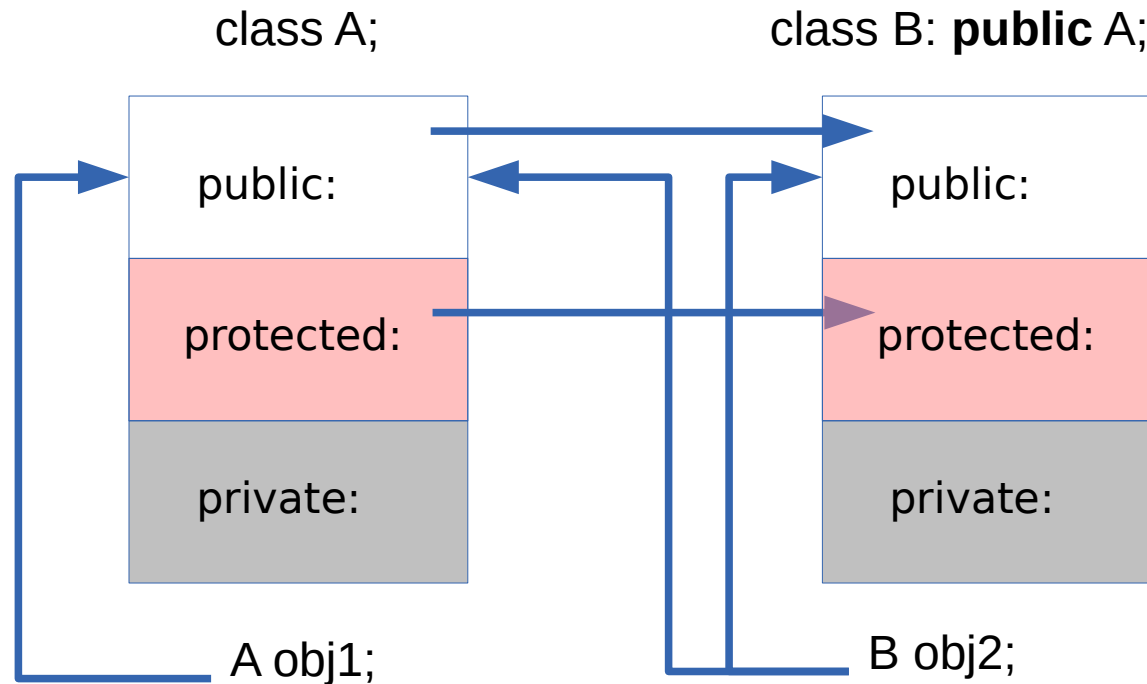
Herencia private

```
class Stack: private List
```

Ejemplo:

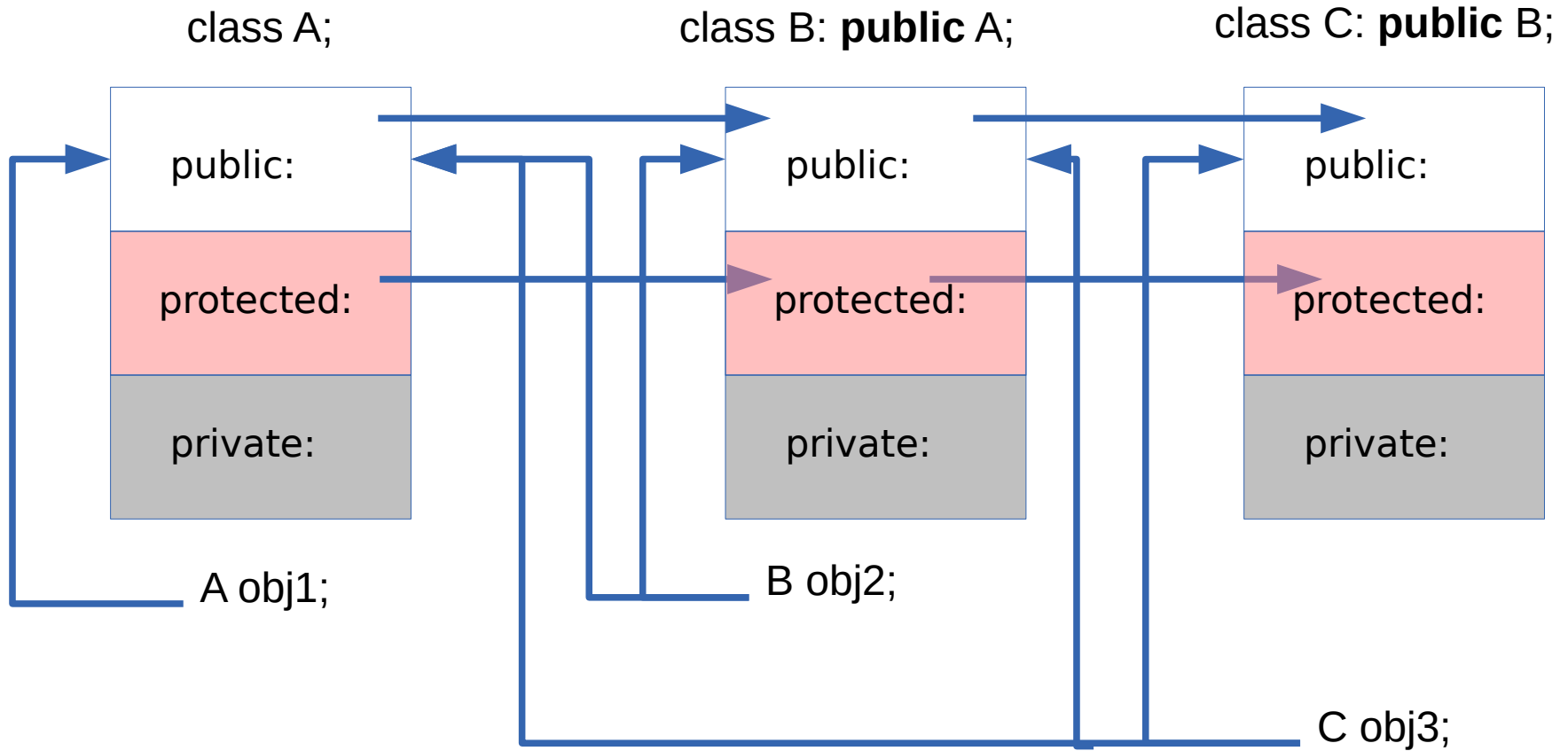
- Una pila “**NO ES**” una lista
- Una pila se crea “**POR MEDIO DE**” una lista

La sección “protected” de una clase:

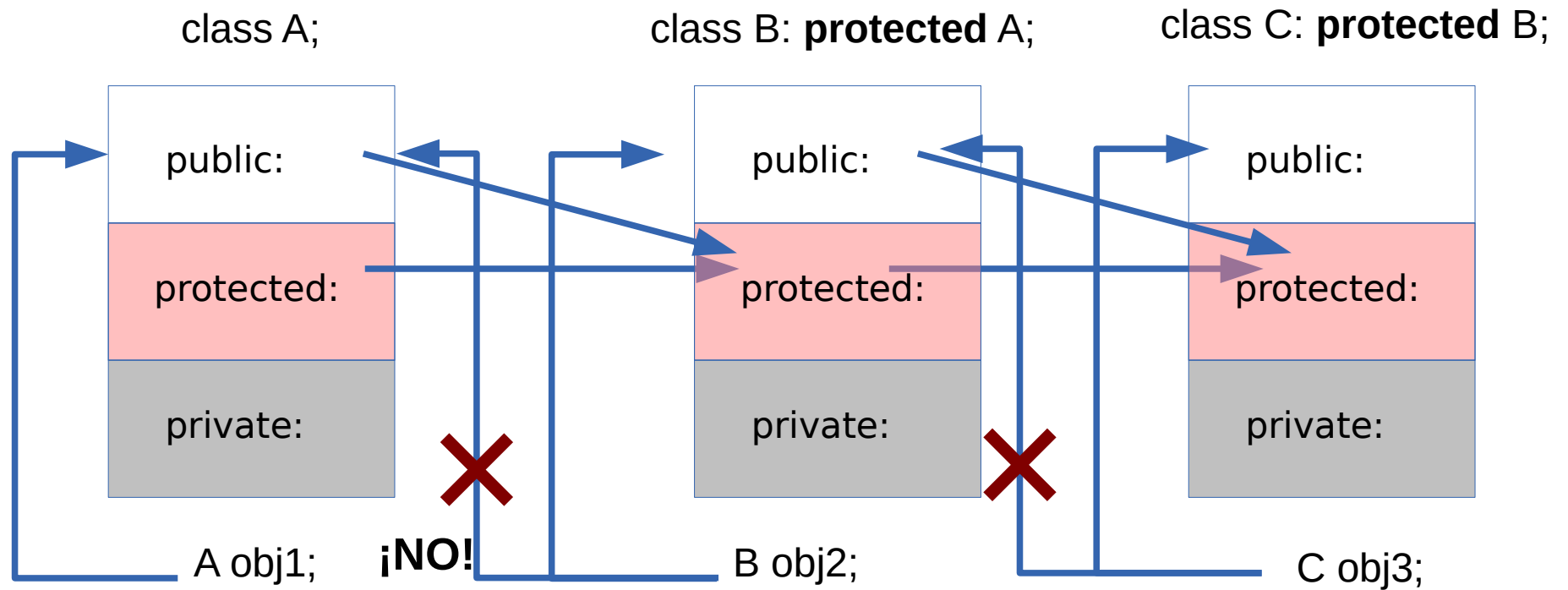


- El desarrollador/a de la clase derivada puede acceder a la parte “protected” de la clase base.
- El desarrollador/a de la clase base permite que las clases derivadas accedan a la sección “protected”.

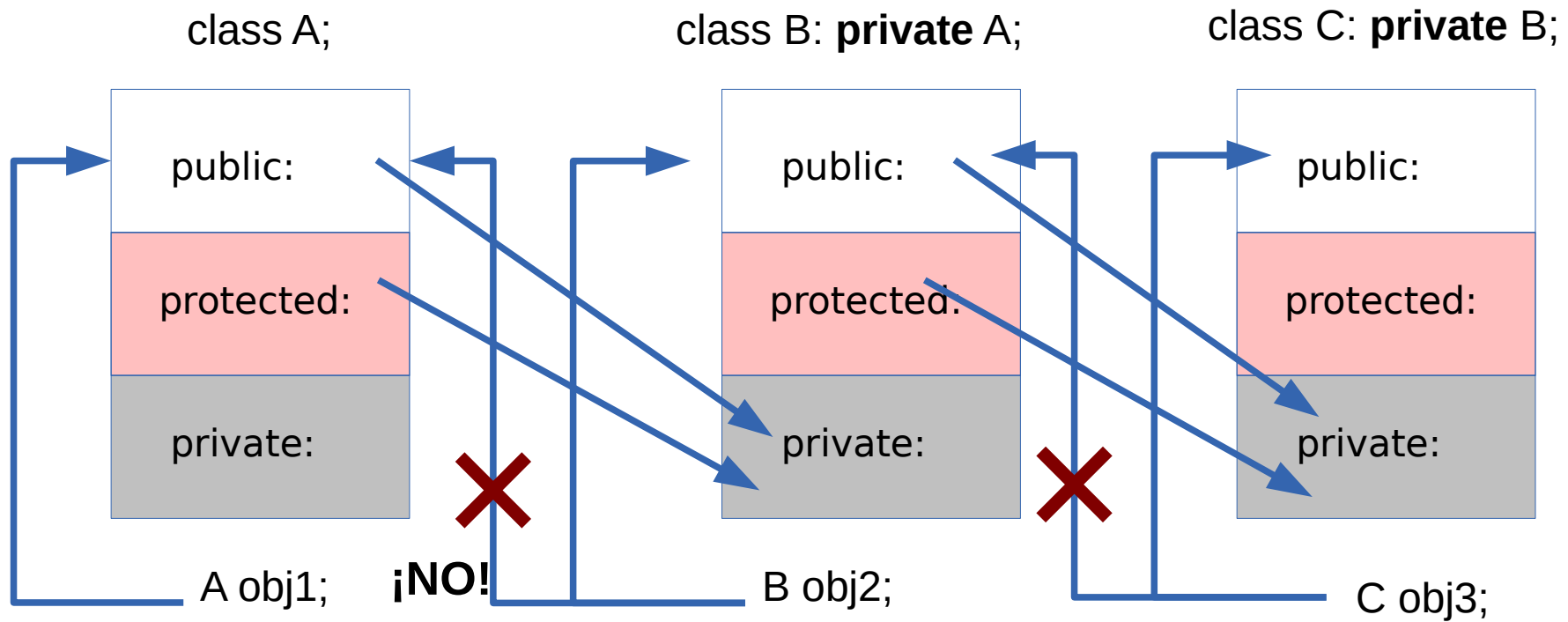
Tipos de herencia: public



Tipos de herencia: protected



Tipos de herencia: private



Herencia public, protected y private

```
class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};

class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};
```

```
class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A // 'private' is
                    // default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

IMPORTANT NOTE: Classes B, C and D all contain the variables x, y and z. It is just question of access.

Fuente: stackoverflow
<https://stackoverflow.com/a/1372858>

Herencia múltiple

```
class D: public A, private B,  
protected C{  
  
    ...  
  
};
```

Objetos pasados por valor (como cualquier variable)

```
class X
{
private:
    int a_;
public:
    X(){a_=5;}; // tres funciones en linea
    int Set(int i){a_=i;};
    int Get(){return a_;};
};

void f(X obj){obj.Set(8);}

int main(void)
{
    X x;
    cout << x.Get();
    f(x);
    cout << x.Get();
}
```

```
$ g++ prueba.cc -o prueba
$ ./prueba
5
5
```

Punteros a objetos

```
Fecha f;  
Fecha *p;  
p=&f;  
cout << p->GetDay(); // f.GetDay()
```

O bien:

```
Fecha *p;  
p= new Fecha(1,1,1970);  
cout << p->GetDay();
```

Polimorfismo en t. de ejecución. Funciones virtuales

```
class Figura{  
protected:  
    double x_, y_;  
public:  
    void SetDim(double i, double j)  
        {x_=i;y_=j;}  
    virtual double Area(){cout << "no  
        definida aqui";return 0.0;}  
};  
  
class Triangulo : public Figura{  
public:  
    double Area() override {  
        // x_ = base, y_ = altura  
        return(x_*y_/2);  
    }  
};  
  
class Cuadrado : public Figura{  
public:  
    double Area() override {  
        // x_, y_ lados  
        return(x_*y_);  
    }  
};
```

Figura es una
clase abstracta

```
int main(void)  
{  
    int opcion;  
    Figura *p;  
    cout << "elige figura \n"  
    << "1.- triangulo \n"  
    << "2.- cuadrado \n";  
    cin >> opcion;  
    if(opcion==1)  
    {  
        Triangulo t;  
        p=&t;  
    }  
    else  
    {  
        Cuadrado c;  
        p=&c;  
    }  
    p->SetDim(3.0,4.0);  
    cout << "\n Area = " << p->Area() <<  
    endl;  
}
```

puntero a
clase base

vinculación
dinámica

Funciones virtuales (clases abstractas)

```
class Figura{
protected:
    double x_, y_;
public:
    virtual double Area()=0;
    virtual double Pinta()=0;
    virtual double Borra()=0;
    virtual double Color()=0;
    virtual double Mueve()=0;
    virtual double Escala()=0;
};
```

```
class Triangulo : public Figura{
public:
    virtual double Area(){...};
    virtual double Pinta(){...};
    virtual double Borra(){...};
    virtual double Color(){...};
    virtual double Mueve(){...};
    virtual double Escala(){...};
    ...
};
```

```
class Cuadrado : public Figura{
public:
    virtual double Area(){...};
    virtual double Pinta(){...};
    virtual double Borra(){...};
    virtual double Color(){...};
    virtual double Mueve(){...};
    virtual double Escala(){...};
};
...
```

Figura es una
clase abstracta

función
polimórfica

```
int ResaltaFigura(Figura *p)
{
    p->Borra();
    p->Mueve(2);
    p->Escala(100);
    p->Color(77);
    p->Pinta();
}
```

código
genérico

vinculación
dinámica

Funciones virtuales puras

```
virtual tipo nombre_funcion(params...) = 0;
```

Funciones virtuales y virtuales puras se usan en clases abstractas

(en C++ se considera que es clase abstracta solo si tiene funciones virtuales puras):

- Definen la interfaz genérica
- No se pueden definir objetos de estas clases
- Son usadas para derivar de ellas y para el polimorfismo
- Clase Abstracta = ABstract Class = ABC = Interface

Funciones virtuales puras (clases abstractas)

```
#include <iostream>
#include <string>
using namespace std;
class Animal
{
private:
    string name_;
public:
    Animal(string name):name_(name){}
    string GetName(){return name_;}
    virtual string Talk() = 0;
};

class Gato: public Animal
{
public:
    Gato(string name):Animal(name){}
    virtual string Talk(){return "Miau!";}
};

class Perro: public Animal
{
public:
    Perro(string name):Animal(name){}
    virtual string Talk(){return "Guau!";}
};
```

```
int main()
{
    Gato g("milu");
    Perro p("boby");

    cout << g.Talk() << endl;
    cout << p.Talk() << endl;
}
```

Si no se declara Talk() dentro de las clases derivadas:

animal.cc: En la función 'int main()':

animal.cc:44:7: error: no se puede declarar que la variable 'p' sea del tipo abstracto 'Perro'

animal.cc:29:7: nota:

porque las siguientes funciones virtual son puras dentro de 'Perro':

animal.cc:15:25: nota: virtual const char* Animal::Talk()

Ejemplo animal.cc

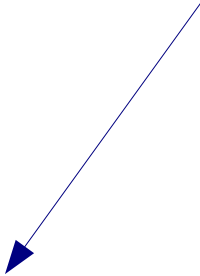
Funciones virtuales puras (clases abstractas)

```
#include <iostream>
#include <string>
using namespace std;
class Animal
{
private:
    string name_;
public:
    Animal(string name):name_(name){}
    string GetName(){return name_;}
    virtual string Talk() = 0;
};

class Gato: public Animal
{
public:
    Gato(string name):Animal(name){}
    virtual string Talk(){return "Miau!";}
};

class Perro: public Animal
{
public:
    Perro(string name):Animal(name){}
    virtual string Talk(){return "Guau!";}
};
```

función
polimórfica



```
void DoTalk(Animal *p)
{
    cout << p->GetName();
    p->Talk();
}
```

Ejemplo animal.cc

Plantillas de función

template
function

```
template <class T>
```

```
void print_vector(T *v, const int n)
{
    for(int i=0;i<n;i++)
        cout << v[i] << " , ";
}
```

```
int main(void)
{
    int a[5]={1,3,5,7,9};
    float b[4]={5.6, 7.8, 3.9, 1.2};
    char c[5]="hola";
    cout << "vector de enteros";
    print_vector(a, 5);
    cout << "vector de floats";
    print_vector(b, 4);
    cout << "vector de char";
    print_vector(c, 4);
}
```

- “T” parámetro formal (tipo genérico)
- Se usa class o typename
- Puede haber varios tipos genéricos:
 <class T1, class T2>
- Pueden tener cualquier nombre

Plantillas de clase (clases genéricas, *class template*)

Class
template



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

```
template <class T> class MiClase{
private:
    T x_, y_;
public:
    MiClase (T a, T b){ x_=a; y_=b;};
    T Div(){return x_/y_};
};
```

```
int main()
{
    MiClase <int> iobj(10,3);
    MiClase <double> dobj(3.3, 5.5);
    cout << "división entera = " << iobj.Div() << endl;

    cout << "división real = " << dobj.Div() << endl;
}
```

salida:

```
$ ./a.out
```

```
division entera = 3
```

```
division real = 0.6
```

Algorithm library

The algorithms library defines functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements

Ver practica correspondiente

Reserva de memoria dinámica (new y delete)

```
int *v;  
float *f;  
v = new int [10] // vector de 10 enteros  
f = new float [5]; // vector de 5 reales  
delete [] v;  
delete [] f;
```

Objetos:

```
A *obj;  
obj = new A(parametros del constructor);  
delete obj;
```

E/S en C++. Streams

(ver apuntes)

E/S en C++. Ficheros

(ver apuntes)

Insertador (<<) y extractor (>>)

```
#include <iostream>
using namespace std;
class Punto{
private:
    int coordx_;
    int coordy_;
public:
    Punto(){coordx_=coordy_=1;};
    friend ostream &operator<<(ostream
&stream, const Punto &p);
    friend istream &operator>>(istream
&stream, Punto &p);
    ...
};
ostream &operator<<(ostream &stream,
const Punto &p)
{
    stream << "(";
    stream << p.coordx_;
    stream << ", ";
    stream << p.coordy_;
    stream << ")";
    return stream;
}
```

extrator
propio

insertador
propio

```
istream &operator>>(istream &stream,
Punto &p)
{
    cout << "Introduce x ";
    stream >> p.coordx_;
    cout << "Introduce y ";
    stream >> p.coordy_;
    return stream;
}
int main(void)
{
    Punto a,b;
    cin >> a;
    cin >> b;
    cout << a << b << endl;
}
```

```
Introduce x 4
Introduce y 1
Introduce x 2
Introduce y 3
(4, 1) (2, 3)
```

Excepciones

```
try{...} // código a monitorizar  
catch (int &i) {...}  
catch (float &i) {...}  
catch (...) {...}
```

- El código que se **monitoriza** (`try`) puede lanzar excepciones de varios tipos que se capturan (`catch`). Cada `catch` es un: *exception handler*.
- Ejemplos en moodle: `ejemplo1Ex.cc`, `ejemplo2Ex.cc`, `ejemplo3Ex.cc` y `ejemploEx4.cc`

Smart Pointers

<https://www.internalpointers.com/post/beginner-s-look-smart-pointers-modern-c>

Every dynamically allocated object (i.e. `new T`) must be followed by a manual deallocation (i.e. `delete T`).

- En el caso de matrices dinámicas, se complica la reserva y también la liberación de la memoria utilizada.
- A partir de C++11, los smart pointers (`unique_ptr`, `shared_ptr` y `weak_ptr`) mejoran estos problemas.
- Fueron ideados para facilitar el trabajo con punteros a objetos

Smart Pointers

```
#include <memory>
```

```
std::unique_ptr
```

- Apunta a un objeto y ningún otro puntero puede apuntar al mismo objeto.
- Cuando se sale del ámbito (out of scope) se elimina el objeto y es liberada su memoria (delete).

```
std::unique_ptr<Type> p(new Type);
```

O bien

```
std::unique_ptr<Type> p = std::make_unique<Type>();
```

FIN

Más . . .

Referencias:

- <http://en.cppreference.com/w/>
- <http://www.cplusplus.com/>
- Bibliografía de la asignatura