Tema 5: Clases y objetos. Espacios de nombres.

Concepto de clase en POO.

Introducción.

- Ya conocemos lo que es una clase del tema anterior.
- También se puede ver como un patrón o una idea de la cual podemos tener realizaciones concretas: *los objetos*.
- Una clase debería tener una única responsabilidad (principio de responsabilidad única).
- Dado que es una buena idea crear una clase por cada responsabilidad detectada, es normal que en un diseño orientado a objetos tengamos que usar varias de ellas.

Forma canónica de una clase.

- Con esta afirmación nos referimos al conjunto de métodos mínimo que debería tener una clase creada por nosotros para que se *pareciera* en su comportamiento lo máximo posible a un tipo de datos básico del lenguaje (*POD*: Plain Old Data).
- También se le llama *interfaz mínimo* de una clase.
- Es interesante que consultes las conocidas reglas de los 3, los 5 y 0.
- Varía de unos autores a otros, incluso de una clase a otra, pero es bastante común uno como el siguiente:

```
class T {
public:
 T  ();                          // 1. Const. por defecto
 T  (const T& t);                // 2. Const. de copia
```

```
T t1, t2;                       // Const. por defecto
T* t3 = new T(t1);              // Const. de copia

t1 = t2;                        // Op. de asignación
...
if (t1 == *t3)...               // Op. de comparación
```

- Es muy útil definir el constructor de copia en los casos donde hay que distinguir entre copia *superficial* y copia *profunda* (shallow/deep copy) ya que el constructor de copia que por defecto nos proporciona el lenguaje, sólo hace una copia *bit a bit* del objeto origen en el objeto destino.
- Al redefinir el operador de asignación en una clase debemos prestar especial atención a esta situación:

```
T t1;
t1 = t1;
```

- ¿Por qué el operador de asignación devuelve un dato de tipo `T&` ?
- Presta atención, no es lo mismo:

```
T t1, t2;
t2 = t1;
```

- Que...

```
T t1;
T t2 = t1;  // Equivale a T t2(t1)
```

- Si fuera necesario podemos añadir operaciones de conversión de tipo desde objetos de la clase a otros tipos:

```
class T {
public:
```

```
  ...
}
```

Hablando de conversión de tipos...

- **C++** soporta la sintáxis de **C** pero introduce (*y aconseja el uso*) una nueva forma de llevarla a cabo.
- Disponemos de una serie de operadores nuevos de conversión de tipo:
  - `static_cast<new_type>(expression)` :
    - Es una conversión de tipo hecha en tiempo de compilación.

```
int n = static_cast<int>(4.3);
```

  - `const_cast<new_type>(expression)` :

```
int main () {
  const int * p = new int(3);
  int* q;

  //q = p; <-- Error!
  q = const_cast<int*>(p); // Ok ahora

  return 0;
}
```

  - `reinterpret_cast<new_type>(expression)` :
    - Obligamos al compilador a que nos haga caso.

```
int main () {
  int * p = new int(3);
  void* q;

  q = p; // Ok
```

```
}
```

- `dynamic_cast<new_type>(expression)` :
  - Se usa en tiempo de ejecución en lugar de en tiempo de compilación.
  - Comprueba si podemos hacer un *downcast*.

```
class B {};
class D : public B {};
D  d;              // Un D
B* b = &d;         // upcast, dynamic_cast may be used,
                  // but unnecessary
D* new_d = dynamic_cast<D*>(b); // downcast
```

Creación de objetos.

- Para crear un objeto basta con crear una variable de esa clase, con el constructor que queramos de entre los que nos proporciona la clase.
- C++ permite crear objetos en:
  - el almacenamiento global

```
MyClass obj;
int main () {...}
```

  - en la pila (*stack*)

```
void f () { MyClass obj;... }
```

  - en el almacenamiento dinámico (*heap*)

```
MyClass* objptr = new MyClass;
```

- Tradicionalmente los LOO crean los objetos en el *heap*. Esto tiene ciertas implicaciones:
  - *La variable es una referencia al objeto.*

almacenamiento global y asignamos uno a otro (caso particular: *troceado* o *slicing*, ocurre cuando asignamos un objeto *completo* de clase derivada a uno de clase base).

Tipos de clases.

- El concepto de *clase* que hemos visto hasta ahora es el que se conoce como **clase concreta**. Podemos crear *instancias* de ella (*objetos*), es decir, se puede *instanciar*.
- Existen *clases que no se pueden instanciar* se llaman **abstractas**. Una *clase abstracta* en **C++** es aquella que tiene al menos un *método abstracto*, en otros LOO basta con que se etiquete como `abstract`.
- En **C++** un *método abstracto* es aquel que no tiene implementación y eso se indica así:

```
class T {                        // T es una clase abstracta
  public:
    T ();                              // Const. por defecto

    virtual void doSomething () = 0; // Método abstracto,
                                       // debe tener enlace dinámico!
}
```

- Si no podemos crear objetos de una *clase abstracta*, ¿cuál es su finalidad?:
  - Actuar como *clase base* de otras que deberán proporcionar una implementación para los métodos abstractos.
- Una clase cuyos métodos son todos *abstractos* y **no** tiene *datos*, se conoce como un **interface**.

Metaclase : Es una clase cuyas instancias son... *clases*!
C++ no tiene metaclases.

Clase final : Es una clase de la que no se puede derivar. **C++** las soporta desde **C++11**.

Local : En los LOO que las soportan, son aquellas declaradas dentro de un función, por tanto sólo se pueden usar allí...¿sólo?

```
  return fT;
}

int main ()  {
   auto n = 2;
   auto frt = f ();

   return frt.n;
}
```

Interna : Se trata de una clase declarada de forma anidada dentro de otra. Según el LOO que usemos, tienen sus peculiaridades.

Parcial : Se trata de aquellas clases cuya declaración se puede extender a lo largo de varios archivos. **C++** no las soporta, **C#** sí.

Clases y zonas de visibilidad en C++.

- **C++** nos proporciona tres zonas de visibilidad para los contenidos de una *clase* o *struct*:
    - public
    - private
    - protected
- **public** : Es visible desde el exterior.
- **private** : Sólo es visible desde *métodos de instancia* o de *clase* de la propia clase.
- **protected** : Sólo es visible desde *métodos de instancia* o de *clase* de la propia clase y de clases derivadas.
- En una *clase* la visibilidad por defecto es *private* y en un *struct* es *public*.

Identificación de clases en un diseño.

1. Escribir una descripción del problema en lenguaje natural.
2. Subrayar los nombres en un color y los verbos en otro.
3. A partir de este momento:
    - Los *nombres* representan objetos (y por tanto *posibles clases*) en nuestro diseño.
    - Los *verbos* representan mensajes que se envían entre los objetos (y por tanto posibles *métodos de clases*).

Evidentemente, el resultado obtenido debe revisarse, juzgarse y adaptar convenientemente al problema que estemos resolviendo.

Espacios de nombres.

El concepto de *espacio de nombres* está relacionado con el de *ámbito*.

- En cierto modo ya lo habeis usado desde *Programación I*, p.e.:

  ```
  void f () { int n; ...}          // No hay colisión con el nombre de la
  void g () { int n; ...}          // variable 'n' igual en ambas funciones
  ```

  Cada función tiene su propio ámbito, su propio *espacio de nombres*.

- Una *clase* o un *struct* también tienen su propio ámbito y por tanto son un *espacio de nombres*.

- Como se puede ver, la finalidad de un *espacio de nombres* es evitar **colisiones** de identificadores (de variables, funciones, clases, etc...) idénticos cuando se emplean en un mismo ámbito.

Espacios de nombres y C++.

- **C++** nos da soporte sintáctico para crearlos, pero aunque el lenguaje de programación que usamos no lo tenga, siempre podemos emplear un *truco*...

```
ListPtr  listCreate   ();         // Espacio de nombres list
Position listLocate   (ListPtr l, Element x);
NodePtr  listRetrieve (ListPtr l, Position i);
...
TreePtr  treeCreate  ();        // Espacio de nombres tree
void     treeDestroy (TreePtr t);
NodePtr& treeRoot    (TreePtr t);
...
GraphPtr graphCreate   ();      // Espacio de nombres graph
void     graphDestroy  (GraphPtr g);
void     graphMakeNull (GraphPtr g);
...
```

La construcción sintáctica en **C++** es la siguiente:

```
namespace NOMBRE {
  class A {
    void m();
    ...
  };
  int n;
  void A::m() { ... }
  void f () {...}
  namespace INTERNO {
    ...
  }
} // <- No lleva ';' al final
```

- Se parece mucho a una *clase*...pero no tiene parte publica, ni privada ni protegida. *Todo es visible desde el exterior*, pero dentro de su ámbito.
- Además, a diferencia de una *clase*, una vez *cerrado* puede volver a abrirse para seguir añadiéndole más identificadores.
- **C++** introduce un operador nuevo para resolver ámbitos: **::**

```
NodePtr& treeRoot    (TreePtr t);
```

- Se definirían y usarían así:

tree.h

```
namespace tree {
  ...
  struct Tree {
    NodePtr _root;
  };
  typedef Tree* TreePtr;
  ...
  TreePtr  create  ();
  void     destroy (TreePtr t);
  NodePtr& root    (TreePtr t);
}
```

tree.cc

```
#include "tree.h"

tree::TreePtr  tree::create  () {...}
void           tree::destroy (tree::TreePtr t) {...}
tree::NodePtr& tree::root    (tree::TreePtr t) {...}
```

- Hay un modo de *sacar* a la vez de su *espacio de nombres* todos los identificadores que define -como te imaginarás esto puede desencadenar colisiones de nombres, *úsalo con juicio*-:

```
namespace tree {...}
using namespace tree;
```

- También es posible extraer de un *espacio de nombres* sólo un símbolo:

```
#include <iostream>
```

- Todos los símbolos que define la *biblioteca estándar* de **C++** se encuentran dentro del espacio de nombres **std**.
- De ahí que, p.e., usemos: `std::string`, `std::cout`, `std::endl`, etc...
- Si en lugar de un *espacio de nombres* se tratara de una *clase*, sería parecido:

tree.h

```
class Tree {
 public:
   typedef Tree* TreePtr;
   ...
   TreePtr  create  ();
   void     destroy (TreePtr t);
   NodePtr& root    (TreePtr t);
 private:
   ...
};

// tree.cc
#include "tree.h"

Tree::TreePtr  Tree::create  () {...}
void           Tree::destroy (Tree::TreePtr t) {...}
Tree::NodePtr& Tree::root    (Tree::TreePtr t) {...}
```

- Podemos emplear los *espacios de nombres* para tener distintas versiones de nuestro código y elegir cual queremos usar p.e. mediante opciones dadas al compilador.
- Un ejemplo de lo anterior lo podemos observar en el siguiente vídeo. En el se propone una posible mejora en un código inicial y para comprobarla se duplica el código. Tanto el código original como el duplicado con la mejora se separan en espacios de nombres distintos.

E/S en C++ bajo el paradigma orientado a objetos.

- **ifstream ::** *streams* de entrada.
- **ofstream ::** *streams* de salida.
- **fstream ::** *streams* de entrada/salida.

Ejemplo de salida.

- Modo por defecto: sobreescribe contenidos:

```
#include <iostream>
#include <fstream>              // cabecera con las declaraciones
using namespace std;

int main () {
  ofstream outputfile;
  outputfile.open ("libro.txt");
  outputfile << "Escribimos en el archivo.\n";
  outputfile.close();
  return 0;
}
```

- El método `open (filename, mode);` permite especificar el modo en el que abrimos el archivo: `ios::in` , `ios::out` , `ios::binary` , `ios::ate` , `ios::app` , `ios::trunc` :

```
ofstream data;
data.open ("data.bin", ios::out | ios::app | ios::binary);
```

Otras operaciones con streams.

- Cierre: `theFile.close()`
- Lectura de lineas:

```
string line;
ifstream theFile ("book.txt");
```

```
    theFile.close();
} else cout << "Hubo un problema al abrir el archivo.";
```

- Estado:
  - `if (myfile.is_open()) { }`
  - `bad()` , `fail()` , `eof()` , `good()`

Archivos binarios.

- Podemos escribir y leer en ellos con `read` y `write` :

```
char* memblock = new char [size];
...
theFile.write ( memory_block, size );
theFile.read  ( memory_block, size );
```

cin, cout y cerr.

- Son *streams*, de entrada el primero y de salida los dos segundos.
- Cualquier función que lea o muestre información de la entrada o en la salida estándar puede, muy fácilmente, hacerlo en un fichero.
- Basta cambiar *cin*, *cout* por la variable de tipo *ifstream* o *ofstream* que usemos en nuestro código.

Redefinición del operador \(<\) \(<\).

- Podemos simplificar el mostrar la información de un objeto en un *stream* de salida redefiniendo como función amiga de su clase este operador, p.e.:

```
#include <iostream>

class Robot {
 public:
  Robot(std::string n, float bl = 1.0) {
    theName = n;
```

```
  virtual ~Robot() {}

private:
  std::string theName;
  float theBattLevel;
};
```

```
// definimos operator<< , se podria hacer dentro de la clase
std::ostream& operator<< (std::ostream& os, const Robot& r) {
  os << "    Name: " << r.theName
     << "\n    BL: " << r.theBattLevel << '\n';
  return os;    //<-- ¿por qué se devuelve este valor?
}

int main(int argc, char *argv[])
{
  Robot r1 = Robot("R1", 0.75);
  Robot r2 = Robot("R2", 0.2);
  Robot r3 = Robot("R3", 0.5);

  std::cout << r1 << r2;
  std::cout << r3;

  return 0;
}
```

Ejecución de una aplicación escrita con un LOO.

- De todo lo descrito hasta ahora es *sencillo* intuir que la ejecución de una aplicación escrita con un LOO y siguiendo los principios de la POO consiste en:

    1. Crear en la función `main` los objetos iniciales de las clases pertinentes de nuestro diseño.

a su vez, enviarán mensajes a otros objetos...y así sucesivamente.

4. El resultado final de todo este paso de mensajes es la *ejecución* de nuestra aplicación.

Ponte a prueba...

```
// Design Patterns: Elements of Reusable Object-Oriented Software
typedef float Coord;
using namespace std;

class Point {
public:
   static const Point Zero;       // Point (0, 0);

   Point (Coord x = 0.0, Coord y = 0.0);

   Coord X () const;  void X (Coord x);
   Coord Y () const;  void Y (Coord y);

   friend Point operator+ (const Point&, const Point&);
   friend Point operator- (const Point&, const Point&);
   friend Point operator* (const Point&, const Point&);
   friend Point operator/ (const Point&, const Point&);
```

```
   Point& operator+= (const Point&);
   Point& operator-= (const Point&);
   Point& operator*= (const Point&);
   Point& operator/= (const Point&);

   Point  operator- ();

   friend bool operator== (const Point&, const Point&);
   friend bool operator!= (const Point&, const Point&);

   friend ostream& operator<< (ostream&, const Point&);
   friend istream& operator>> (istream&, Point&);
```

```
const Point Point::zero = Point(0,0);
```

...una vez más!

```
// Design Patterns: Elements of Reusable Object-Oriented Software
class Rect {
public:
   static const Rect Zero;        // Rect (Point (0, 0), Point (0, 0));

   Rect (Coord x, Coord y, Coord w, Coord h);
   Rect (const Point& origin, const Point& extent);

   Coord Width  () const; void Width  (Coord);
   Coord Height () const; void Height (Coord);
   Coord Left   () const; void Left   (Coord);
   Coord Bottom () const; void Bottom (Coord);
```

```
   Point& Origin () const; void Origin (const Point&);
   Point& Extent () const; void Extent (const Point&);

   void MoveTo (const Point&);
   void MoveBy (const Point&);

   bool IsEmpty  () const;
   bool Contains (const Point&) const;
};
```

¿Quieres probar con variables y métodos de clase?

```
  void show_vehicle_clount_address();

private:
  string brand;
  static uint32_t vehicle_count;
```

```
// Reserve space for class variable.
uint32_t Vehicle::vehicle_count = 0;

Vehicle::Vehicle(const string& br) {
  brand = br;
  vehicle_count++;

  cout << "Created vehicle: " << brand << " [" << this << "]\n";
}

Vehicle::~Vehicle() {
  cout << "Destroyed vehicle: " << brand << " [" << this << "]\n";
}

uint32_t Vehicle::get_number_of_vehicles() {
  // Prueba a descomentar esta linea:
  // cout << "Asking number of vehicles for this = " << this << '\n';
  return vehicle_count;
}

const string &Vehicle::brand_name() {
  cout << "Asking for brand-name of [" << this << "]\n";

  // return brand;
  //     ^^^^^ is a shorthand for:
  return this->brand;
}

void Vehicle::set_brand_name(const string &brand) {
  // this-> is mandatory here.
  this->brand = brand;
}

void Vehicle::show_vehicle_clount_address() {
  cout << "For vehicle " << this->brand
  << " its vehicle_count address is @: ["
  << &Vehicle::vehicle_count << "]\n";
}
```

```
{
  Vehicle s("Seat"), a("Audi"), m("Mercedes");

  cout << "\nTotal vehicles created-a: "
  << Vehicle::get_number_of_vehicles()
  << '\n';

  // Works too in C++!
  cout << "Total vehicles created-b: "
  << s.get_number_of_vehicles()
  << "\n\n";

  // Getting brands:
  cout << "s: " << s.brand_name () << '\n';
  cout << "a: " << a.brand_name () << '\n';
  cout << "m: " << m.brand_name () << '\n';
  cout << "\n";

  // Change seat's brand:
  s.set_brand_name("Cupra");
  cout << "Changed seat's brand to: " << s.brand_name () << '\n';
  cout << "\n";

  cout << "Show vehicle_counter static-var address-of each vehicle:\n";
  s.show_vehicle_clount_address();
  a.show_vehicle_clount_address();
  m.show_vehicle_clount_address();
  cout << '\n';

  return 0;
}
```

Amistades...¿peligrosas?

Las funciones *amigas* de una clase:

- No son métodos de la clase.

- Por tanto, elige bien a qué funciones/métodos les *otorgas* el calificativo de *amigas*.

```
class X {
 private:
    int a;

    // Declaración de amistad
    friend void friendSetA(X& xr, int i);
 public:
    // Es un método de instancia
    void memberSet(int i) { a = i; }
};

// Es amiga de la clase X
// Por eso puede acceder a 'a'.
void friendSetA(X& xr, int i) {
  xr.a = i;
}
```

En relación con la amistad, ten en cuenta que:

- La amistad **no se hereda**: *los amigos de una clase base no son amigos de una clase derivada*.
- La amistad **no es transitiva**: *los amigos de mis amigos no son mis amigos*.

La biblioteca estándar de C++.

¿Qué es?

- Se trata de un conjunto de estructuras de datos y algoritmos predefinidos que *vienen* de manera estándar junto al compilador del lenguaje.
- Estos elementos que proporciona están definidos en archivos de cabecera. No es necesario enlazar con un archivo binario para su uso en la gran mayoría de casos.
- Para evitar colisiones de nombres todos los identificadores se encuentran bajo el *espacio de nombres* `std` .

- Para poder usar los elementos que nos proporciona debemos incluir *normalmente* un solo archivo de cabecera, p.e.:
  - `<array>` , `<vector>` , `<queue>` , etc...
  - `<algorithm>`

Para saber más sobre la biblioteca estándar.

- Echa un vistazo a este vídeo para saber algo más sobre alguno los *contenedores* empleados más habitualmente.
- Y si con la biblioteca estándar no tienes suficiente, mira todo lo que ofrece boost.

Aclaraciones.

- **Este contenido no es la bibliografía completa de la asignatura**, por lo tanto debes estudiar, aclarar y ampliar los conceptos que en ellas encuentres empleando los enlaces web y bibliografía recomendada que puedes consultar en la página web de la ficha de la asignatura y en la web propia de la asignatura.