

# Programación Orientada a Objetos. Curso 2023-24

## PRÁCTICA 1. C++: clases, objetos, funciones inline, espacio de nombres.

### Introducción a CMake.

#### NOTAS INICIO DE PRÁCTICAS:

- Crea una carpeta que se llame 'poo' para las prácticas de esta asignatura. Mete todos los ejercicios de esta práctica 1 en el subdirectorio poo/p1. Haz lo mismo para el resto de prácticas cambiando el número de cada una.
- Para hacer las prácticas en cualquier sistema operativo debemos tener instalado: Make, CMake, gcc, g++, gdb y git. Y un editor de código fuente, para lo que recomendamos Visual Studio Code de Microsoft disponible para Windows, Linux, macOS y Web.

**1.- Clases y Objetos.** Para definir una clase en C++ se utiliza la palabra reservada 'class'. En el siguiente código se declara la clase 'Person' (en C++ la doble barra '//' es un comentario hasta el final de esa línea):

```
// Declaración de la clase Person
#include <string>
class Person{
    private:
        std::string name_;
        int age_;
        double rank_;
    public:
        Person(std::string name, int age, double rank) {name_=name;age_=age;rank_=rank;}
        std::string GetName(){return name_;}
        int GetAge(){return age_;}
        double GetRank(){return rank_;}
};
```

Fíjate que en C++, para usar la clase std::string, debes incluir la cabecera <string> (que no termina en .h).

Escribe la clase 'Person' en el fichero 'person.h' (recuerda usar guardas de inclusión siempre en los ficheros de cabecera) y escribe la función 'main()' en un fichero aparte que se llame 'person-main.cc' que cree varios objetos de tipo 'Person' y saque por pantalla sus datos con instrucciones como las siguientes. Dentro de la función 'main()', para crear 2 objetos (p1 y p2) de tipo 'Person';

```
Person p1("Juan", 32, 4.568), p2("Ana", 41, 7.371);
```

Los parámetros que se pasan en estas declaraciones irán a una función especial que se llama igual que la clase sin tipo devuelto, y que recibe el nombre de constructor de la clase.

Para acceder al nombre del objeto p1 y p2 se invoca a la función correspondiente de la forma:

```
p1.GetName(); // Juan
p2.GetName(); // Ana
```

En C++ hay un objeto que representa la pantalla:

```
std::cout
```

Para usar el objeto std::cout tendrás que incluir la correspondiente cabecera:

```
#include <iostream>
```

Como el objeto cout está dentro del espacio de nombres std (más adelante veremos qué es un espacio de nombres) debes utilizarlo con el operador '::' (*scope resolution operator*) así:

```
std::cout
```

que hace referencia al objeto 'cout' dentro del espacio de nombres 'std'. Igual que en la clase 'Person', la declaración 'std::string' hace referencia a la clase 'string' dentro del espacio de nombres 'std'.

Y para enviarle la cadena a visualizar en pantalla se utiliza el operador de inserción (*insertion operator*):

```
<<
```

Por tanto la línea quedará:

```
std::cout << p1.GetName();
```

Para *autodocumentar* mejor tus ficheros de código, recuerda a partir de ahora incluir al inicio de cada fichero 2 líneas al principio con el nombre del archivo y un breve comentario descriptivo (mejor si es en inglés). Por ejemplo, en este caso sería:

```
// person-main.cc  
// A example program with a simple class
```

NOTA: la extensión habitual para archivos con código C++ es ".cc", pero hay otras como ".C", ".cpp", ".++", ".c++", ".cxx" y otras. **Nosotros usaremos la extensión ".cc" para el código y ".h" para las cabeceras.**

**2.- Compilando con CMake: *the standard build system for C++*.** Para compilar nuestro programa vamos a ayudarnos de la utilidad CMake (<https://cmake.org/>) que es una herramienta para automatizar el proceso de compilación de un proyecto (build automation).

Compilar programas en C++ es diferente según el sistema operativo (Windows, Linux, mac, etc.), según el compilador que se use, según la versión del compilador, etc. Y hay muchas opciones de compilación. Y si además hay muchos ficheros, todo puede ser un lío... CMake nos lo facilita todo. CMake lo que nos hace es crear el Makefile a partir de unas sencillas declaraciones que hacemos en el fichero CMakeLists.txt de cada proyecto.

Para ello crea en el mismo directorio p1 un fichero que se llame CMakeLists.txt con el siguiente contenido:

```
cmake_minimum_required(VERSION 3.10)  
project(Práctica1)  
add_executable(person-main person-main.cc)
```

Cuando se compila un proyecto de cierto tamaño, se compilan muchos ficheros en distintos directorios y se generan ficheros objeto, librerías, ejecutables, etc. Todo lo que se genera en la compilación conviene tenerlo separado en otro directorio diferente al del código fuente. Esto es lo que se llama *out-of-source builds*.

Nosotros vamos a crear en nuestros proyectos un subdirectorio que se llame 'build' donde compilaremos y obtendremos los ficheros objeto y ejecutables.

Entonces, dentro del directorio 'p1' creamos el directorio 'build':

```
$ mkdir build  
$ cd build
```

Invocamos CMake siempre dentro del directorio 'build'. Y como el fichero CMakeLists.txt principal (el top-level CmakeLists.txt) está en el directorio superior hacemos:

```
$ cmake ..
```

Si no hay errores, se habrá generado el Makefile que será lo que utilizemos a partir de este momento cuando queramos compilar. OJO: se compila con make, CMake es únicamente para generar el Makefile. Una vez generado el Makefile, ya siempre compilamos únicamente con make y siempre dentro del directorio 'build':

```
$ make
```

Comprobamos que se ha creado el ejecutable y lo invocamos:

```
$ ./person-main
```

Cambia el código de person-main.cc para que saque por pantalla todos los datos del objeto p1 y del objeto p2, es decir, nombre, edad y rango de cada uno, cada uno en una línea diferente.

### 3.- Indicar la versión del compilador de C++ con CMake

Vamos añadir una nueva función a la clase Person que nos saque todos los datos del objeto a la vez:

```
std::string GetDataStr(){
    return name_ + " " +
           std::to_string(age_) +
           " " + std::to_string(rank_) + "\n";
}
```

Esta función la añadimos al resto en el fichero person.h.

Al compilar de nuevo ejecutando:

```
$ make
```

Seguramente nos dará un error la función std::to\_string() ya que esta función se incluye a partir del estándar C++11 (esta función convierte una variable a string).

Para añadir el estándar que queremos usar al compilar debemos añadir a nuestro CMakeLists.txt las siguientes líneas antes de la llamada a la función project():

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)
```

En este caso, no solo establecemos C++17 como nuestro estándar para compilar, sino que lo fijamos como requisito, el cuál ya incluye al estándar C++11 que es el que necesitamos.

Al haber cambiado CMakeLists.txt necesitamos ejecutar de nuevo cmake:

```
$ cd build
$ cmake ..
$ make
```

Ahora el programa compilará sin errores.

### 4.- Los espacios de nombres (namespaces) en C++.

Un espacio de nombres (namespace) es un bloque de código entre llaves con definiciones en su interior. Estas definiciones quedan asociadas al nombre de dicho *namespace*. Dos espacios de nombres diferentes

pueden definir el mismo identificador sin que exista conflicto. Como en este ejemplo:

```
namespace ns1{
int a; // Esta es la variable ns1::a
int b; // Esta es la variable ns1::b
}

namespace ns2{
float a; // Esta es la variable ns2::a
float c; // Esta es la variable ns2::c
}
```

Así, la variable `ns1::a` será diferente a la variable `ns2::a`, y podemos hacer:

```
#include <iostream>
// Aquí irían las declaraciones de ns1 y ns2
int main(void)
{
int a=55;
ns1::a=0;
ns2::a=2.3;
std::cout<< "ns1::a= " << ns1::a << "\n";
std::cout<< "ns2::a= " << ns2::a << "\n";
std::cout<< "a= " << a << "\n";
}
```

De esta forma podemos crear un espacio de nombres sin preocuparnos de que otro programador utilice los mismos identificadores que nosotros. Ten en cuenta que esto es muy útil en grandes proyectos software para que no se den conflictos de nombres. Eso sí, no puede haber dos espacios de nombres con el mismo identificador en un mismo proyecto.

Añade estos espacios de nombres antes de la función `main()` en el fichero `person-main.cc` y añade a la función `main()` todo el código dentro de la función `main()` anterior. Compila todo y comprueba su resultado.

Debes entender bien el concepto de espacio de nombres, y que

```
ns1::a
ns2::a
a
```

son tres variables con el mismo nombre ‘a’, pero son diferentes y perfectamente usables sin conflicto gracias a los espacios de nombres.

“std” es un gran espacio de nombres (el espacio de nombres estándar) ya definido internamente por C++ y donde se encuentran montones de objetos y clases que usaremos en nuestros programas.

En vez de estar repitiendo todo el rato ‘std::’ delante de cada clase o función del espacio de nombres std, puedes usar la instrucción:

```
using namespace std;
```

y ya no habría que poner delante `std::`: ya que esta instrucción establece el espacio de nombres por defecto en el que buscar (esto se podría hacer con cualquier espacio de nombres, por ejemplo `ns1` o `ns2` ).

Cambia la función `main()` y quita ‘std::’ de todas las instrucciones.

Esto nos ahorra código en algunos casos, pero no es conveniente porque puede generar confusiones. Por ejemplo imagina que hacemos:

```
using namespace ns1;  
using namespace ns2;
```

Ahora, si nos referimos a la variable 'a': ¿a cuál nos referiríamos?, ¿la de ns1 o la de ns2?... tendríamos un conflicto de nombres.

Siempre es mejor usar la expresión larga ns1::a o ns2::a o std::cout o std::string que usar *using namespace* que genera confusión y conflictos de nombres.

Quita todas las instrucciones using namespace del código y añade std:: o el espacio de nombres necesario en cada caso.