

# Programación Orientada a Objetos. Curso 2019-20

## Práctica 1. Primer contacto con C++ y Unit Testing

1.- Hacer el típico programa “hello world” en C++ en el fichero `hello-world.cc`. Para escribir en pantalla deberás usar el objeto `cout`, para el que tendrás que incluir la correspondiente cabecera

```
#include <iostream> //fíjate que no termina en .h
```

(En C++ puedes añadir comentarios con `//` hasta el final de esa línea)

Cada vez que uses el objeto `cout` tendrás que indicar que se encuentra asociado al espacio de nombres `std` de la siguiente forma:

```
std::cout << "hello world\n"; // cout visualizará en pantalla los datos
                        // que reciba detrás del operador <<
                        // Se pueden encadenar varios datos añadiendo nuevos <<
```

NOTAS:

`<<` es un operador (*insertion operator*) que envía o inserta datos en este caso a `cout`.

`::` es un operador (*scope resolution operator*) que se usa para asociar elementos, en este caso el objeto `cout` al espacio de nombres `std`.

Un espacio de nombres (*namespace*) es un bloque de código C++ con definiciones en su interior asociadas a dicho namespace. Dos namespaces diferentes pueden usar el mismo identificador sin que exista conflicto, así la variable `namespace1::a` será diferente a la variable `namespace2::a`.

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

```
// hello-world.cc
// A program that prints the immortal saying "hello world"
```

—  
Completa el código en el fichero fuente `hello-world.cc` que contendrá la función `main()` y compila el programa con `g++` corrigiendo posibles errores.

NOTA: la extensión habitual para archivos con código C++ es “.cc”, pero hay otras como “.C”, “.cpp”, “.++”, “.c++”, y otras. Nosotros usaremos la extensión “.cc”

2.- Hacer un programa (`guess.cc`) que genere un número aleatorio entre 1 y 10, y solicite al usuario un número por teclado para posteriormente adivinarlo indicando al usuario si el número generado es menor, mayor o es correcto. Usar para la entrada por teclado el objeto de entrada estandar `cin` para lo que tendrás que incluir la misma cabecera que con `cout` del ejercicio anterior. En el siguiente

código leemos una variable entera desde el teclado y la metemos en “i”:

```
std::cout << "Introduce un número: ";  
std::cin >> i;
```

Recuerda que en la generación de números aleatorios, o mejor dicho pseudoaleatorios en este caso (¿sabes lo que es un número pseudoaleatorio?, ¿y una semilla? ... haz una pequeña investigación), la semilla se puede establecer mediante:

```
srand(time(NULL)); // time(NULL) returns "UNIX Epoch", this is seconds  
// since 00:00:00 1 January 1970
```

y luego con `rand()` podrás generar un número entre 0 y `RAND_MAX` (¿sabes lo que es `RAND_MAX`? Haz una pequeña investigación. Ver enlace en moodle).

Tendrás que incluir las cabeceras `cstdlib` y `ctime` para `srand()` y `time()` respectivamente:

```
#include <cstdlib> // Fíjate que no llevan la terminación .h  
#include <ctime>  // Fíjate que no llevan la terminación .h
```

Son cabeceras también usadas en C, que ahora en C++ se usan anteponiendo una “c” delante y sin la terminación “.h”.

A partir de ahora usa siempre los objetos `cin` y `cout` para entrada/salida teclado/pantalla.

Los objetos `cin` y `cout` son propiamente “streams”, es decir, cadenas que se asocian con operaciones de lectura o escritura a ficheros. En este caso *cin* es el fichero asociado a la entrada (el teclado) y *cout* a la salida (la pantalla).

### 3.- Introducción a las clases. La clase *Dados*.

En Programación Orientada a Objetos, una clase es una plantilla extensible para la creación de objetos con los datos que necesiten esos objetos y sus operaciones asociadas. En este ejercicio vamos a codificar la clase *Dados*.

Hay juegos de azar que usan 2 dados para poder jugar. Codificaremos la clase *Dados* que representará dichos dados, por lo tanto la clase *Dados* tendrá 2 variables enteras (int) para guardar los valores de cada uno de los 2 dados. Los datos de una clase irán normalmente en la sección “private” de esa clase.

Además añadiremos una serie de operaciones a la clase. Las operaciones (funciones o métodos) van normalmente en la sección “public” de la clase.

La declaración de la clase *Dados* debe ir en el fichero `dados.h` y el cuerpo de sus funciones en el fichero `dados.cc`

Métodos de la clase *Dados*:

- lanzamiento: obtiene un nuevo valor aleatorio para los dos dados

- `getDado1`: devuelve el valor del primer dado
- `getDado2`: devuelve el valor del segundo dado
- `setDado1`: recibe un entero como único parámetro, y lo asigna al dado 1 si dicho entero está entre 1 y 6, en cuyo caso devuelve `true`. En caso contrario no lo asigna y devuelve `false`.
- `setDado2`: igual a la anterior para el segundo dado.
- `getSuma`: devuelve el valor de la suma de los dos dados.
- Constructor. Un constructor es un método público de la clase que se llama exactamente igual que la clase, que no devuelve nada (ni siquiera `void`) y que se ejecuta automáticamente al declarar un objeto de esa clase. En el caso de la clase `Dados`, su constructor debe establecer la semilla de la generación de números aleatorios y asignar un valor inicial a los dados igual a 1.

Puedes guiarte por el siguiente código y ampliarlo para incluir en él el resto de métodos de la clase `Dados`:

```
// dados.h
// La clase dados representa el lanzamiento de 2 dados
class Dados{
private:
    int d1_;
    int d2_;
public:
    Dados();           // el constructor no devuelve nada
    int getDado1();    // el método devuelve un entero
    . . .
};
```

```
// dados.cc
// Cuerpo de los métodos de la clase Dados
Dados::Dados(){
    . . .
}
int Dados::getDado1(){
    . . .
}
```

Crear un fichero `juego.cc` con la función `main()` que declare un objeto de la clase *Dados*, y que permita invocar todas y cada una de las operaciones de la clase `Dados` mediante un sencillo menú.

```
//juego.cc
// Programa principal prueba funcionamiento de la clase Dados
#include . . .
. . .
int main(){
    Dados d;
    . . .
```

```
std::cout << "dado 1 = " << d.getDado1();
. . .
}
```

Analizar todos los `#includes` necesarios para el programa y **usar guardas de inclusión siempre** en nuestros ficheros `.h` (¿sabes lo que son guardas de inclusión?.. lee una entrada específica que hay en *moodle* que lo explica)

Usar la guía de estilo que se ha dejado en el *moodle* de la asignatura para nombrar, clases, funciones, variables, etc. (Google C++ Style Guide). Por ejemplo, esta guía nos recomienda añadir al final del identificador de una variable privada un subrayado “\_” (*underscore*).

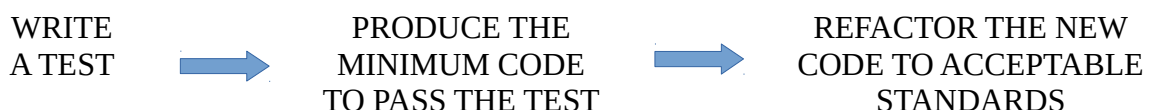
Como ejercicio adicional, intentar acceder directamente a los datos privados de la clase *Dados* desde la función `main()`. Por ejemplo añadiendo al final de la función `main` un `cout` que saque en pantalla el valor del primer dado mediante el objeto de tipo *Dados*, el operador punto y el nombre del dato privado. Compilar el programa y analizar el error que indica el compilador en el siguiente código:

```
std::cout << d.d1_;
```

C++ no nos deja acceder directamente a los datos privados de una clase, el compilador nos da un error. Éstos solo pueden accederse desde los métodos de dicha clase. En clase de teoría veremos el porqué.

4.- Introducción a las pruebas unitarias (**UNIT TESTING**). Introducción al desarrollo mediante la elaboración de tests o desarrollo guiado por tests o *Test-Driven Development* (TDD):.

Comenta la siguiente figura:



Refactoring: “*Code refactoring is the process of restructuring existing computer code – changing the factoring – without changing its external behavior. Refactoring improves nonfunctional attributes of the software. Advantages include improved code readability and reduced complexity to improve source code maintainability, and create a more expressive internal architecture or object model to improve extensibility.*” ([http://en.wikipedia.org/wiki/Code\\_refactoring](http://en.wikipedia.org/wiki/Code_refactoring))

(“*Factoring*”: reconsiderando, reajustando, reestructurando...)

Nosotros vamos a usar **Google Test**, un framework considerado de tipo **xUnit**. xUnit es una familia de frameworks para pruebas unitarias (*unit testing*) que comparten características comunes que veremos en esta práctica.

Ejercicio:

- a) En el moodle hay un enlace a googletest en GitHub. Entra y pulsa en “releases” y busca la versión v1.8.1 (es la última estable). Descarga el zip del código fuente y lo guardas en tu directorio HOME. Descomprime el zip con unzip en tu directorio HOME (se habrá creado un directorio con esa versión de googletest). Entra dentro de dicho directorio. Solo nos interesa la carpeta “googletest” de ese directorio. Mueve dicha carpeta “googletest” a el directorio donde vas a tener las prácticas de POO. Si este directorio se llama “~/poo” tendrías que ejecutar:

```
mv googletest/ ~/poo
```

- b) Descargar del moodle de la asignatura el archivo *datos\_unittest.zip* y descomprimirlo en el directorio donde tenéis la clase *Dados*. Verás los ficheros *datos\_unittest.cc* y *Makefile*
- c) Adapta el Makefile a tu proyecto. Para esto solo hay que cambiar el valor de la variable GTEST\_DIR y ponerle el *path* a donde hayas copiado la carpeta *googletest*.
- d) Una vez hecho eso, ejecutar:

```
make
```

- e) Abrir el fichero *datos\_unittest.cc* y analizar cada uno de los test que se han incluido en él. Para ello se puede consultar el ejemplo en la documentación de googletest a través del enlace proporcionado a su documentación. En unit testing, el *test runner* es el programa que ejecuta los diferentes tests diseñados y muestra los resultados, en este caso el programa *datos\_unittest.cc*
- f) Ejecutar los tests sobre la clase *Dados* mediante el fichero ejecutable creado con el makefile, ejecutando:

```
./datos_unittest
```

- g) Analizar cada línea de la salida de la ejecución de los test.
- h) Vamos a utilizar este tipo de test a lo largo de todo el curso. Para familiarizarnos mejor con ellos podemos introducir algún error en cada una de las operaciones de la clase *datos* y ejecutar los test. Ir corrigiendo cada error, uno a uno y ejecutando los tests en cada corrección, hasta que se corrijan todos los errores.
- i) Añade un test para la clase *Dados* al fichero *datos\_unittest.cc* para que compruebe el buen funcionamiento de una operación que se llame “getDiferencia” y que devuelva un entero con la distancia entre el menor y el mayor valor de los almacenados en los *datos*.

- j) Después de añadir el test escribe el código de dicha operación dentro de la clase de forma que pasen los test que has creado para ella.
- k) Por último limpia y optimiza el código de toda la clase *Dados* para que quede al máximo nivel de calidad, autodocumentación, etc. Y vuelve a ejecutar los test para confirmar que todo ha quedado correcto.

**5.- Añadir las siguientes operaciones a la clase *Dados* :**

- a) El método *getLanzamientos1()* devolverá el número de veces que se ha realizado el lanzamiento del dado 1. El método *getLanzamientos2()* será igual para el dado 2. Para ello la clase *Dados* debe contabilizar cada lanzamiento. Tener en cuenta que cada vez que se ejecute el método *setDado1()* se debe contabilizar un nuevo lanzamiento del dado 1. Igual pasa con *setDado2()* y el dado 2. Por lo tanto el número de lanzamientos del dado 1 no tiene por qué coincidir con el número de veces que se ha lanzado el dado 2. Además, tener en cuenta que cada vez que se ejecute el método *lanzamiento()* se debe incrementar el número de lanzamientos de ambos dados.
- b) Añade un test que compruebe que justo después de declarar un objeto de la clase *Dados* los métodos *getLanzamientos1()* y *getLanzamientos2()* devuelven 0. Y que después de un número determinado de lanzamientos *n*, los métodos devuelven el valor *n*.
- c) Añadir los métodos *getMedia1()* y *getMedia2()* que devolverán la media de los valores que van saliendo en el dado 1 y 2 respectivamente. Los valores iniciales (igual a 1) de los dados que se fijan en el constructor no deben ser considerados para la media. Si se pide la media antes del primer lanzamiento, ésta debe ser cero (añadir un test que compruebe esto).
- d) Si se usa *setDado1()* o *setDado2()* el valor asignado debe ser considerado para la media (añadir un test que compruebe esto).
- e) Añade el método *getUltimos1()* que debe recibir un vector de enteros como parámetro al que se le deben copiar los 5 últimos valores obtenidos para el primer dado, de modo que el último valor del dado quedará en la posición 0 del vector, el penúltimo en la posición 1, etc.. El método *getUltimos2()* se comporta de forma análoga para el segundo dado. Por tanto, debes modificar la clase *Dados* para que pueda ir almacenando los 5 últimos lanzamientos de esta forma y añade los tests que comprueben su funcionamiento correcto.

**6.-** En moodle, dentro de la sección dedicada a la práctica 1 verás “PRUEBA FINAL”. Se trata de un test (*dados\_unittest.cc*) que debes copiar a tu carpeta de trabajo (antes de eso, haz una copia con otro nombre del fichero *dados\_unittest.cc* que has estado usando en el ejercicio anterior). Este archivo tiene 10 tests realizados por el profesor de la asignatura que deberán ejecutarse todos correctamente para validar tu trabajo definitivamente.

**7.-** Echa un vistazo y estudia todo el material y todos los enlaces que se han dejado en el moodle de la asignatura correspondientes a esta práctica. Verás que también

hay una sección con material adicional y enlaces de interés en el moodle de la asignatura que debes revisar de vez en cuando para familiarizarte con todo ello.