Programación Orientada a Objetos. Curso 2021-22
Práctica 1. Primer contacto con C++, las clases y el unit testing

1.- Comencemos con el típico programa "hello world" en C++ que escribiremos en el fichero:

`hello-world.cc`

En C++, para escribir en pantalla la cadena "hello world" deberás usar el objeto:

`cout`

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

```
#include <iostream>  // fíjate que no termina en .h
        // Doble barra o slash comenta hasta el final de la línea
```

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

`std`

Como el objeto `cout` está dentro del espacio de nombres `std` debes utilizarlo con el operador `::` (*scope resolution operator*) así:

`std::cout`

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

`<<`

Por tanto la línea quedará:

```
std::cout << "hello world\n";
```

En el ejercicio 2 entenderás mejor el concepto de espacio de nombres.

Para autodocumentar mejor tus ficheros de código, recuerda a partir de ahora incluir al inicio de cada fichero 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 inmortal saying "hello world"
```

Completa el código en el fichero `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. En esta asignatura usaremos la extensión ".cc"

2.- 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 n1{
int a; // Esta es la variable n1::a
int b; // Esta es la variable n1::b
```

```
}

namespace n2{
int a;  // Esta es la variable n2::a
int c;  // Esta es la variable n2::c
}
```

Así, la variable `n1::a` será diferente a la variable `n2::a`, y podemos hacer:

```
int main(void)
{
int a=55;
n1::a=0;
n2::a=2;
std::cout<< “n1::a= “ << n1::a << “\n”;
std::cout<< “n2::a= ” << n2::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. Mientras los nombres de los propios espacios de nombres sean diferentes, no habrá problema. Ten en cuenta que esto es muy útil en grandes proyectos software.

Como ejemplo escribe un programa que se llame `ns.cc` que declare los dos espacios de nombres anteriores y que en su función `main()` ejecute el código del ejemplo anterior.

Reflexiona y comenta con tus compañeros/as este código. Debes entender bien el concepto de espacio de nombres, y que

```
n1::a
n2::a
  a
```

son tres variables diferentes perfectamente usables sin conflicto gracias a los espacios de nombres.

Una vez hayas compilado y ejecutado el ejemplo anterior cambia el programa `ns.cc` y añade (en la siguiente línea al *#include*) la siguiente instrucción para utilizar `std` como namespace por defecto:

```
using namespace std;
```

Cambia la función `main()` y observa que ahora no es necesario poner `std::` delante del objeto `cout` (aunque, como veremos en clase de teoría, es preferible usar siempre la forma larga `std::cout`).

3.- Hacer un programa en C++ (`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 cada vez 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 y usar el operador de extracción (*extraction operator*):

```
>>
```

En el siguiente código leemos una variable entera desde el teclado y la almacenamos 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
```

Fíjate en que 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` en tus programas para entrada/salida teclado/pantalla.

Los objetos `cin` y `cout` son técnicamente “*streams*”. Un *stream* es una secuencia de objetos, en este caso de caracteres. La secuencia `cin` está asociado a la entrada (el teclado) y `cout` a la salida (la pantalla).

4.- Introducción al concepto de “clase”. 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 las operaciones asociadas a dichos objetos. En este ejercicio vamos a codificar una clase en C++, la clase *Dados*.

En C++ declarar una clase es muy sencillo, se hace mediante un bloque de código donde se declaran variables y funciones.

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.

Dentro de una clase hay tres secciones: private, public y protected. La parte *private* es para uso interno del programador que está escribiendo la clase, aquí es donde declaramos las variables (los datos) que necesitamos.

La parte *public* de una clase es para las funciones (u operaciones, o métodos) de la clase y pueden ser llamadas desde fuera de una clase, por eso son públicas. La parte public es la interfaz de la clase.

La parte *protected* la veremos más avanzado el curso.

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

Escribiremos los siguientes métodos o funciones en la parte *public* 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, la función 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 lanzamiendo 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. Observa en el siguiente ejemplo cómo se declara un objeto de la clase *Dados* y como se invoca a la función `getDado1()` con el operador punto con dicho objeto.

```
//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:

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

Compila el programa y analiza el error que indica el compilador.

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 nos extenderemos más en ello.

5.- 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)

(*"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 busca la última versión (latest) en versiones (releases). Actualmente la última versión estble es la v1.11.0. En la página releases, abajo en Assets pulsa para descargar el "Source code (zip)" y lo guardas en tu directorio HOME. Descompríme 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 *dados_unittest.zip* y descomprimirlo en el directorio donde tenéis la clase *Dados.* Verás los ficheros *dados_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 dados_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 `dados_unittest.cc`
f) Ejecutar los tests sobre la clase *Dados* mediante el fichero ejecutable creado con el makefile, ejecutando:

```
./dados_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 dados 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 dados_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 dados.
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.

6.- 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 e 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.

7.- 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.

8.- 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.