Sesión 2

Especificación y uso de módulos en C++ (I)

2.1 La clase `Estudiante`

En esta sesión mostraremos ejemplos en C++ de **especificación** y **uso** de módulos. Veremos como la metodología del diseño modular es fácilmente aplicable a los lenguajes orientados a objetos, como C++. El contenido de esta sesión está basado en la primera parte del *documento introductorio a la orientación a objetos*, disponible en la web de la asignatura.

Un **módulo de datos** contiene un tipo y sus operaciones. En C++ obtendremos una *clase* y sus *métodos*. Aunque cambian los nombres, los conceptos son los mismos. Este paso suele requerir un cambio en la parametrización de las operaciones, bien por las razones expuestas en la sesión anterior o bien para aprovechar las facilidades de C++ para la programación orientada a objetos, en la que cada objeto es el "propietario" de sus métodos.

En cuanto al programa principal, el criterio no varía respecto a los programas de un sólo módulo. El cuerpo del programa dará lugar al método `main` y si hacen falta operaciones adicionales, éstas se traducirán normalmente.

Consideremos el ejemplo del módulo **Estudiante**. Al traducirlo a C++ hemos obtenido la correspondiente *clase*. Podéis consultar los detalles en el fichero `Estudiante.doc`. Además de las operaciones ya conocidas, hemos incluido sendos métodos de lectura y escritura de estudiantes.

2.1.1 El parámetro implícito

Notad que hemos cambiado la parametrización de las operaciones, de modo que tanto las modificadoras como las consultoras ya no poseen un parámetro de tipo `Estudiante` sino que operan sobre un **objeto implícito** de la clase `Estudiante`. Dicho objeto se concreta al usar las operaciones.

Ejemplo: la consultora `tiene_nota` pasa de esta cabecera en la notación algorítmica

```
funcion tiene_nota (e: Estudiante) dev b: booleano
```

a esta otra en C++

```
bool tiene_nota () {
   bool b;
   ...
   return b;
}
```

Dado un estudiante `e`, la primera se aplica así

```
b:=tiene_nota(e);
```

mientras que la segunda se aplica así

```
b=e.tiene_nota();
```

2.1.2 Ejemplo: Redondeo de la nota de un estudiante

En el fichero `red1.cpp` tenemos un programa que permite sustituir la nota de un estudiante por su correspondiente redondeo con resolución 0.1 más próximo, mediante la fórmula:

```
nota_red = ((int)(10*(nota+.05)))/10.;
```

La sustitución puede realizarse mediante una acción o una función. Comprobad las diferencias entre ambas versiones.

2.1.3 Puesta a punto del programa `red1.cpp`

Para usar la clase `Estudiante` hemos creado tres ficheros.

- `Estudiante2.hpp`: contiene el nombre de la clase y las cabeceras de sus operaciones, amén de otros elementos que el compilador necesita para compilar cualquier programa que utilice la clase. Observad que hemos incluido este fichero al principio de `red1.cpp`. Lo hemos instalado en `/assig/prap/cplus/includes`.
- `Estudiante2.cpp`: contiene la implementación de todas las operaciones y todos los elementos necesarios para generar el fichero objeto de la clase. Hemos compilado este fichero para obtener el correspondiente fichero objeto de la clase y ya no lo necesitamos más. Por eso, de momento no está disponible.
- `Estudiante2.o`: el mencionado fichero objeto. Deberéis enlazarlo con el fichero `red1.o`, para obtener el `red1.exe`. Lo hemos instalado en `/assig/prap/cplus/objetos` y hemos definido la variable `OBJETOS_CPP` para abreviar dicha ruta.

El proceso completo para obtener el ejecutable `red1.exe` será:

```
g++ -c red1.cpp -I$INCLUDES_CPP  --> genera red1.o

g++ -o red1.exe red1.o $OBJETOS_CPP/Estudiante2.o  --> genera red1.exe
```

2.1.4 Ejercicio: Redondeo de la nota de una secuencia de estudiantes

Modificad el programa anterior para redondear las notas de una secuencia de estudiantes. Usad como marca de final de secuencia un estudiante cuyo DNI tiene valor 0. Un ejemplo de entrada de datos está en el fichero `datosredsec.txt`

Emplead siguiente esquema en el método `main`

```
leer_estudiante
mientras no ultimo_estudiante hacer
   redondear_estudiante
   escribir_estudiante
   leer_estudiante
fmientras
```

2.1.5 Ejercicio: Redondeo de la nota de un vector de estudiantes

Modificad el programa anterior, de forma que el redondeo se realice mediante una acción que actúe sobre un vector de estudiantes. Escribid operaciones de lectura y escritura para el vector, basadas en el ejemplo `busqueda_lin` de la sesión anterior, de forma que el programa principal se reduzca a las declaraciones correspondientes más las llamadas a dichas operaciones:

```
llamada a leer_vector
llamada a redondear_vector
llamada a escribir_vector
```

2.1.6 Ejercicio: Búsqueda en un vector de estudiantes

Modificad el programa `busqueda_lin` de la sesión anterior para que, dados un vector de estudiantes y un DNI, compruebe si existe el estudiante correspondiente y retorne su nota, si la tiene. El resultado esperado puede ser uno de los siguientes:

```
El estudiante no esta en el vector

El estudiante esta en el vector, pero no tiene nota

El estudiante esta en el vector y su nota es <la que sea>
```

Podéis emplear cualquiera de las técnicas introducidas en el ejercicio de donde procede el programa `busqueda_lin` para obtener un resultado booleano que indique si el estudiante está o no en el vector, y un resultado entero con la posición en la que está, en caso de éxito. A partir de dicha posición podréis realizar el análisis de la nota.

2.1.7 El operador de asignación

Escribid un programa que demuestre que, con la presente implementación de la clase `Estudiante`, la asignación de estudiantes realmente actúa como una copia. Concretamente, comprobad que después de realizar una asignación entre dos estudiantes, cualquier modificación sobre uno de ellos no afecta al otro. ¿Qué sucede si dicha asignación se realiza de forma simultánea con la declaración del segundo estudiante?

2.1.8 Ejercicio: Simplificación de un vector de estudiantes agrupados

Consideremos un vector de estudiantes no vacío que puede contener estudiantes repetidos (con el mismo DNI, aunque la nota puede variar). Además, suponemos que todas las apariciones de un mismo estudiante son consecutivas. Programad una función que obtenga un segundo vector donde cada cada estudiante sólo aparezca una vez, con la nota más alta de ese estudiante en el vector original. El vector resultado ha de conservar el orden de los estudiantes en el vector original. Intentad aprovechar la propiedad de que todas las apariciones de un mismo estudiante en el vector son consecutivas para no realizar cálculos innecesarios. En particular, cada posición del vector original solo debe visitarse una vez y una vez ocupada cada posición del vector resultado, no deben visitarse las anteriores.

Ejemplo: si el vector original es (recordad que las notas no válidas dan lugar a un estudiante sin nota)

```
4444 7.3  4444 7.6  2222 -1  2222 5.1  3333 14
```

el vector resultante debería ser

```
4444 7.6  2222 5.1  3333 NP
```

Probad vuestro programa en situaciones extremas como:

- Un vector con todos los estudiantes distintos
- Un vector con todos los estudiantes iguales (mismo DNI)
- Un estudiante aparece con varias notas y su nota máxima es la primera
- Un estudiante aparece con varias notas y su nota máxima es la última
- Un vector con un solo elemento
- ...

2.1.9 Ejercicio: Simplificación de un vector de estudiantes sin agrupar

Resolved el problema anterior sin suponer que todas las apariciones de un mismo estudiante son consecutivas. El vector resultado ha de conservar el orden de los estudiantes, respecto a *su primera aparición* en el vector original. El programa resultante será menos eficiente: se ha de conservar la propiedad de visitar sólo una vez cada posición del vector original, pero no podremos evitar repetir visitas en el vector resultado. Los DNI no pueden usarse como índices de vectores.

Ejemplo: si el vector original es (recordad que las notas no válidas dan lugar a un estudiante sin nota)

```
4444 7.3  5555 3.45 2222 -1 4444 7.6  2222 5.1  3333 14
```

el vector resultante debería ser

```
4444 7.6  5555 3.45  2222 5.1  3333 NP
```