Sesión 3

Implementación de módulos en C++

- Este documento contiene ejercicios que hay que resolver en el Jutge (en la lista correspondiente del curso actual) y que aquí están señalados con la palabra *Jutge*.
- Recomendamos resolver los ejercicios en el orden en el que aparecen en este documento. No se supervisarán los problemas del Jutge si antes no se han resuelto los ejercicios previos.

3.1. La clase `Estudiant`

En una sesión anterior introdujimos la clase `Estudiant`. Mostramos su especificación y la manera de usarla en nuestros programas. También comprobamos que su fichero `Estudiant.hh` estaba en la carpeta `/assig/pro2/inclusions` para que el compilador lo encontrase, pues queríamos incluirlo en nuestros programas.

Lo que no mostramos, sin embargo, fue el fichero `Estudiant.cc`. En esta sesión veremos una versión del mismo y aplicaremos algunas modificaciones sobre ella. Una vez terminada cada fase de modificaciones, volveremos a compilar y linkar los programas de la sesión anterior con la clase resultante para comprobar los efectos de dichos cambios.

Cuando se implementa una clase, siempre ha de considerarse que puede tener más de una representación correcta. Normalmente, se comienza proponiendo una representación sencilla, aunque quizás no óptima, y luego se buscan maneras de mejorarla. Lo mismo ocurre con el código de las operaciones. En general, tendremos total libertad para introducir cambios en una clase si no afectan al comportamiento de las operaciones de cara al exterior, es decir, si la nueva versión sigue cumpliendo las especificaciones del tipo. Si por cualquier motivo dicho comportamiento se ve alterado, deberemos hacerlo constar en la documentación correspondiente.

3.1.1. Elementos públicos y privados; elementos visibles y ocultos

Hemos comentado en clase de teoría que la especificación de un módulo ha de ser independiente de su implementación. La especificación de un módulo no debe contener elementos que permitan al usuario de la misma saber cómo está implementada o, en su defecto, no debe permitir al usuario acceder a dichos elementos para consultarlos o modificarlos.

En C++ disponemos de varios mecanismos para conseguir dicha independencia. En primer lugar, habréis visto que en el fichero `Estudiant.hh` hay dos secciones: la declaración de *elementos públicos* y la declaración de *elementos privados*. Usamos la primera para declarar las cabeceras de las operaciones públicas de la clase, de forma que dichas operaciones se puedan utilizar fuera de la misma. En la segunda declaramos los campos de datos y las cabeceras de las operaciones privadas de la clase. Eso impide usar dichos elementos fuera de la misma.

También es posible ocultar los elementos privados de la clase, lo que nos permitiría, por ejemplo, no mostrar los campos de la clase `Estudiant` en el fichero `Estudiant.hh`. Esto supondría un incremento de la independencia de la especificación respecto a la implementación. Sin embargo, las herramientas para hacerlo no forman parte del contenido de esta asignatura. En las clases que vosotros diseñaréis no os pediremos la ocultación de los campos, nos conformaremos con *campos privados visibles*, como los que veremos en esta sesión.

3.1.2. Ejemplo: una primera implementación completa

En la carpeta de la sesión tenéis los ficheros `Estudiant.hh` y `Estudiant.cc`. Si repasamos el primero veremos las dos partes mencionadas: los campos de datos (visibles, pero privados) y las cabeceras de la operaciones de la misma (públicas).

La representación elegida para el tipo `Estudiant` es una lista de campos de tipos simples que consiste en un entero para el DNI, un `double` para la nota y un booleano que indica si el estudiante tiene nota o no. Notad que no es necesario usar `structs` u otros constructores del lenguaje C++.

El código de las operaciones aparece en el fichero `Estudiant.cc`. Notad que las situaciones no permitidas por las precondiciones se controlan explícitamente mediante *excepciones*. No os pediremos que lo hagáis así en vuestros programas (bastará con tener disponibles maneras de comprobar las precondiciones) pero aquí lo incluimos a modo de ejemplo. En cualquier caso, notad que usamos mensajes de excepciones a la medida de cada error.

Observad también que en las cabeceras de las operaciones del fichero `Estudiant.cc` va insertada la declaración `Estudiant::`, para indicar que estas operaciones son las que se han declarado en `Estudiant.hh`. Toda otra operación que encontrásemos en `Estudiant.cc` sería considerada ajena a la clase y no tendría derecho a ver sus campos (tampoco tendría, obviamente, parámetro implícito).

Podéis compilar y linkar el programa `red1.cc` de la sesión anterior con esta clase y comprobaréis que sigue funcionando. Para que se vea que no hay trampa, deberéis linkar el `red1.o` con el `Estudiant.o` de la carpeta de la sesión en lugar del de la ruta `$OBJECTES`. El compilador da prioridad al de la carpeta de la sesión si lo ejecutamos desde ésta.

La secuencia de comandos queda así:

```
p2++ -c red1.cc -I$INCLUSIONS
p2++ -c Estudiant.cc -I$INCLUSIONS
p2++ -o red1.exe red1.o Estudiant.o
```

3.1.3. Ejercicio: una implementación alternativa

Suprimid el campo `amb_nota` de la representación de la clase y dejad solamente `DNI` y `nota`. Revisad todas las operaciones de `Estudiant.cc` que usaban dicho campo para que sigan funcionando correctamente sin él. Ello obliga a modificar el criterio para saber si un alumno tiene nota o no. Utilizad un valor especial del campo `nota` cuya presencia permita deducir que la nota no ha sido añadida aún (ha de ser un valor que no se confunda con una nota válida, por ejemplo el -1). **Importante:** actualizad el invariante de la representación de la clase.

Aprovechad en todo momento la posibilidad de acceder a los campos de la clase, no uséis llamadas a operaciones públicas de la misma.

Notad el uso del `this` para diferenciar el campo `nota` y el parámetro del mismo nombre. Tendréis que usarlo en más instrucciones que en la implementación original.

Compilad el fichero `Estudiant.cc` y comprobad que `red1` sigue funcionando con esta nueva versión sin modificar su código. Es conveniente recompilarlo, ya que la nueva clase tiene un campo menos, aunque las cabeceras de las operaciones no hayan cambiado. Por otra parte, es obligatorio volver a linkarlo con el nuevo `Estudiant.o`, ya que en otro caso no estaremos probando las modificaciones realizadas.

3.1.4. Constructoras y destructoras

Una particularidad de C++ es que si una clase no tiene ninguna creadora explícita, C++ le dota de una creadora por defecto (es este caso sería `Estudiant()`), pero en caso contrario la creadora por defecto no existe. Por ejemplo, si eliminamos `Estudiant()`, cada vez que declaremos un nuevo estudiante sin parámetros el compilador detectará un error, porque no está definida explícitamente.

- Introducid una constructora adicional que permita declarar y copiar estudiantes simultáneamente. Su especificación será

  ```
  Estudiant(const Estudiant& est);
  /* Pre: cert */
  /* Post: El resultado es un estudiante nuevo, copia de est */
  ```

  Recordad que, cuando se programa dentro de una clase, es posible acceder directamente a los campos de los objetos de la misma, no solo a los del parámetro implícito de una operación.

  Por ejemplo, si deseamos asignar el DNI de `est` al parámetro implícito de esta operación, podemos hacer

```
dni = est.consultar_DNI();
```

pero también (de hecho, es preferible)

```
dni = est.dni;
```

Para conseguir que al declarar un nuevo estudiante `est2`, éste se cree como una copia de `est`, invocaremos la operación así

```
// ..... se supone que est1 ya ha sido obtenido

Estudiant est2(est1);
```

- Modificad la función `redondear_e_f` de `red1` que redondea la nota de un estudiante, de modo que el estudiante retornado se cree a partir del estudiante original mediante una llamada como la que acabamos de mostrar.

En cuanto a las operaciones destructoras, C++ se comporta de manera análoga, es decir, si una clase no tiene definida una, se le proporciona por defecto. Por el momento, eso será suficiente para nosotros, pero durante el curso veremos clases que necesitarán que se las programemos.

- Comprobad el funcionamiento de la destructora `~Estudiant()` insertando en su código una instrucción que escriba un mensaje por pantalla. Veréis que al ejecutar el programa `red1` (o cualquier otro), el mensaje se escribe cada vez que se libera el espacio de un estudiante.

3.1.5. Operaciones `static`

Hemos visto en clase de teoría que si una operación de una clase no va a usar su parámetro implícito la declaramos "`static`". Una operación `static` no tiene parámetro implícito. Tiene acceso a los campos `static` de la clase pero el resto de información que puede manejar hay que pasársela como parámetros "normales", es decir, entre paréntesis.

Un ejemplo típico es la operación privada `cerca_dicot` de la clase `Cjt_estudiants`: notad que le pasamos como parámetro el vector del conjunto, el DNI del estudiante buscado y los extremos del intervalo de búsqueda. Si dentro de dicha operación se hiciese referencia a cualquier campo no `static` de la clase, como por ejemplo `nest`, se produciría un error de compilación.

En general, usaremos esta categoría de operaciones para representar las operaciones *privadas* de una clase que no requieran parámetro implícito. Sin embargo, solo veremos unos pocos ejemplos de operaciones *públicas* `static`.

Ejercicio: La operación `comp` del fichero `Cjt_estudiants.cc` es anómala en el sentido que no pertenece a ninguna clase. Eliminadla y convertidla en una operación pública (y `static`) de

la clase Estudiant. Los estudiantes comparados seguirán apareciendo como parámetros "normales" de la operación y el código de ésta podrá conservar las llamadas a consultar_DNI o, mejor, podrá acceder directamente a los respectivos campos DNI de dichos estudiantes.

Notad que al usar ahora comp en la llamada a sort de la operación ordenar_cjt_estudiants obtendremos un error de compilación si no decimos que comp pertenece a la clase Estudiant: eso se hace escribiendo Estudiant::comp.

Para comprobar que lo habéis hecho bien, compilad las nuevas versiones de Estudiant y Cjt_estudiants y usadlas para poner a punto el programa presentats_conj.cc de la carpeta de la sesión. Leed los datos del conjunto sin ordenar por DNI, para comprobar que el conjunto se mantiene ordenado sin problemas.

Recordad que para linkar dicho programa debéis aplicar el siguiente comando (compilad previamente Cjt_estudiants.cc si aún no lo habéis hecho)

```
p2++ -o presentats_conj.exe presentats_conj.o Estudiant.o Cjt_estudiants.o
```

3.2. La clase Cjt_estudiants

Trabajemos ahora con la clase Cjt_estudiants. Modificaremos algunas de las operaciones originales y añadiremos otras nuevas.

3.2.1. Cambios en las especificaciones

- Reprogramad la operación afegir_estudiant para que su precondición sea solamente que el parámetro implícito no esté lleno y ella misma realice la búsqueda correspondiente. Su especificación queda así

```
void afegir_estudiant(const Estudiant& est, bool& b);
  /* Pre: el paràmetre implícit no està ple */
  /* Post: b indica si el paràmetre implícit original conté un estudiant amb el
     dni d'est; si b=fals, s'ha afegit l'estudiant est al paràmetre implícit */
```

- Realizad una manipulación semejante con las operaciones modificar_estudiant y consultar_estudiant. Esta última pasará a ser void: ya no retornará un Estudiant sino que lo obtendrá como parámetro por referencia, junto con el correspondiente booleano.

- Simplificad la solución del último ejercicio de la sesión 2 usando las nuevas versiones de las operaciones modificadas.

3.2.2. Una mejora de eficiencia

Notad que en la operación afegir_estudiant la búsqueda propuesta es la lineal, pero dado que el vector está ordenado, podría usarse la búsqueda dicotómica. De esta forma, la nueva

versión de la operación quedaría optimizada. Notad que la versión habitual de la búsqueda dicotómica no es suficiente, ya que lo que necesitamos aquí es obtener la posición ocupada por el estudiante dato, si está en el conjunto y la que debería ocupar, si no está.

Probad la nueva versión sobre la solución del último ejercicio de la sesión 2.

3.2.3. Ejercicio: Conjunt d'estudiants amb imax (X68173) de la Lista *Sessió 3* (*Jutge*)

Añadid a la clase las operaciones `esborrar_estudiant` y `estudiant_nota_max`, presentadas en los apuntes de teoría. Notad que, al añadir el campo `imax`, una gran parte de las operaciones originales deben modificarse para gestionar dicho campo. Usad el valor -1 para representar la situación de que el conjunto no contenga ningún estudiante con nota y, por tanto, no exista el estudiante de nota máxima. **Importante:** hay que usar el `Cjt_estudiants` original, no el resultante de resolver los ejercicios anteriores.

Para ayudar a depurar vuestra solución, desarrollad un programa para probar las operaciones de este ejercicio. Dicho programa no se ha de entregar.

3.2.4. Ejercicio: Control - Torn 1 (Primavera 2015) (X90633) de la Lista *Sessió 3* (*Jutge*)

Este ejercicio requiere implementar versiones de `afegir_estudiant` y `esborrar_estudiant` con cabeceras similares a las de 3.2.1.