Sesión 2

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

- 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.

2.1. La clase `Estudiant`

En esta sesión veremos ejemplos de **especificación** y **uso** de módulos en C++. El contenido de esta sesión está basado en el documento *Introducció al disseny modular i al disseny basat en objectes*, disponible en la web de la asignatura y usado en las clases de teoría.

Como se explica en dicho documento, la herramienta de C++ que usamos para programar modularmente es la *clase*. Eso lleva consigo, entre otras cosas, un cambio en la parametrización de las operaciones, pues se aplica la filosofía de que cada objeto es el "propietario" de sus métodos.

En cuanto al programa principal, el criterio no variará respecto a los programas de un sólo módulo. Su código contendrá al método `main` y si hacen falta operaciones adicionales, éstas se añadirán normalmente.

Consideremos el ejemplo del módulo `Estudiant` visto en clase de teoría. Podéis consultar su especificación en el fichero `Estudiant.pdf`.

2.1.1. El parámetro implícito

Notad que ni las modificadoras ni las consultoras poseen un parámetro de la clase `Estudiant` sino que operan sobre un **parámetro implícito** de dicha clase. Tal objeto se concreta al usar las operaciones.

Ejemplo: la consultora `te_nota` posee esta cabecera

```
bool te_nota ()
/* Pre: cert */
/* Post: El resultat indica si el paràmetre implícit té nota */
```

Dado un estudiante `e`, dicha operación se aplica así

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

2.1.2. Ejemplo: Redondeo de la nota de un estudiante

En el fichero `red1.cc` tenemos un programa que permite sustituir la nota de un estudiante por su correspondiente redondeo a la décima más próxima, 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. Observad también que esas operaciones están parametrizadas de la forma tradicional, ya que no pertenecen a ninguna clase.

2.1.3. Puesta a punto del programa `red1.cc`

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

- `Estudiant.hh`: 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.cc`. Lo hemos instalado en `/assig/pro2/inclusions` para no copiarlo cada vez que lo necesitemos.
- `Estudiant.cc`: 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.
- `Estudiant.o`: el mencionado fichero objeto. Deberéis enlazarlo con el fichero `red1.o` para obtener el `red1.exe`. Lo hemos instalado en `/assig/pro2/objectes`.

Para compilar `red1.cc` hemos de indicar donde está el fichero `Estudiant.hh`. Para ello, disponemos de la opción `-I`

```
p2++ -c red1.cc -I/assig/pro2/inclusions
```

que genera el fichero `red1.o`. Obsérvese que no se indica el nombre del fichero que necesitamos, sino únicamente el directorio donde está. Si se necesitan varios ficheros que están en el mismo directorio no hay que hacer nada más.

Para linkar escribiríamos

```
p2++ -o red1.exe red1.o /assig/pro2/objectes/Estudiant.o
```

que genera `red1.exe`. En esta caso hay que mencionar todos los ficheros que necesitemos.

Para poder trabajar con más comodidad y usar fácilmente todos los ficheros que iremos necesitando a lo largo del curso, que estarán en los directorios `/assig/pro2/inclusions` y `/assig/pro2/objectes`, es conveniente crear variables de entorno. Para ello escribiremos:

```
setenv INCLUSIONS /assig/pro2/inclusions
setenv OBJECTES /assig/pro2/objectes
```

Si aplicamos el comando `echo`, veremos el valor que han tomado las variables

```
echo $INCLUSIONS  (ha de retornar /assig/pro2/inclusions)
echo $OBJECTES  (ha de retornar /assig/pro2/objectes)
```

Ahora podemos compilar y linkar de una forma más cómoda:

```
p2++ -c red1.cc -I$INCLUSIONS   --> genera red1.o
p2++ -o red1.exe red1.o $OBJECTES/Estudiant.o   --> genera red1.exe
```

Para que las definiciones de `INCLUSIONS` y `OBJECTES` sean permanentes, editad el fichero `.tcshrc` en vuestro **directorio principal** y escribidlas en él. A partir de ahora, las definiciones se aplicarán automáticamente cada vez que abráis una sesión en Linux[1]. Para esta primera ocasión en que la sesión ya está abierta podéis ejecutar

```
source .tcshrc
```

y las definiciones surtirán efecto.

Notad que el nombre del fichero `Estudiant.hh` va entre comillas en la inclusión. La diferencia con las inclusiones anteriores es que cuando se emplean ángulos `<...>` el compilador busca los ficheros en cuestión primero entre los componentes estándar de C++ y después en los introducidos por la opción `-I`. Cuando se introducen entre comillas, los busca primero en el directorio del usuario y, solo si no los encuentra, en los mencionados en el caso anterior. Por eso, mantendremos la política de usar ángulos para las inclusiones de elementos estándar y comillas para los definidos por nosotros.

Por último los ficheros del directorio `OBJECTES` son el resultado de compilar ficheros fuente en el entorno linux64 de la FIB y no necesariamente linkarán con ficheros compilados en otro entorno. Los ficheros fuente correspondientes se pueden encontrar en `OBJECTES/SOS` y se pueden recompilar en otros entornos diferentes.

[1] En realidad, este comando redefine las variables, destruyendo sus valores anteriores, si los tuvieran. Hay que tener cuidado al elegir los nombres de las variables de entorno, para no usar uno que ya exista y sea importante.

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. Guardad el programa resultante en un fichero distinto. Usad como marca de final de secuencia un estudiante cuyo `DNI` tiene valor 0. Un ejemplo de datos y resultados está en el fichero `redsec.dat`.

Emplead el siguiente esquema en el método `main`

```
leer_estudiante
while (not ultimo_estudiante) {
   redondear_estudiante
   escribir_estudiante
   leer_estudiante
}
```

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

Dado un vector de estudiantes, escribid una acción que redondee las notas de sus componentes. Escribid operaciones de lectura y escritura para el vector, basadas en los ejemplos de la sesión anterior. En un fichero llamado `vectorIOEstudiant.hh` poned las cabeceras de las operaciones y en `vectorIOEstudiant.cc` el código. El esquema del programa principal ha de ser:

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

Un ejemplo de datos y resultados está en el fichero `redvec.dat`

Nos interesa que las comprobaciones de acceso a posiciones prohibidas de un vector de objetos se realicen de forma más estricta. Para ello contamos con la opción de compilación `-D_GLIBCXX_DEBUG` ya incluida en `p2++`. No se deben linkar ficheros objeto compilados con esta opción con ficheros compilados sin esta opción. Esta situación no debería darse en nuestra asignatura porque siempre compilaremos con `p2++`.

Comprobad el error de ejecución resultante si en el bucle del redondeo se visita la posición `size()` del 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. Usad también los ficheros `vectorIOEstudiant.hh` y `vectorIOEstudiant.cc`.

Diseñad una operación `busqueda_lin_vest` que realice la búsqueda por `DNI`. El programa principal debe tener la estructura

```
declaraciones
llamada a leer_vector
llamada a busqueda_lin_vest
escritura de resultados
```

El resultado esperado puede ser uno de los siguientes (solo se puede usar `cout` en el `main`):

```
El estudiante no está en el vector

El estudiante está en el vector, pero no tiene nota

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

Podéis emplear cualquier técnica conocida para que la operación `busqueda_lin_vest` obtenga un resultado que permita saber si el DNI buscado está en el vector y, en caso afirmativo, su posición en el mismo. A partir de dicha posición podréis realizar el análisis de la nota para escribir el resultado correcto. Un ejemplo de datos y resultados está en el fichero `busquedalin_vest.dat`.

2.1.7. Ejercicio: Notas máxima y mínima de un vector de estudiantes

Modificad las funciones `max_min` de la sesión anterior para obtener las notas máxima y mínima de entre los estudiantes con nota de un vector, visitando una sola vez cada elemento de éste. Usad también los ficheros `vectorIOEstudiant.hh` y `vectorIOEstudiant.cc`. Podéis suponer que el vector no es vacío y que existe al menos un estudiante con nota. Un ejemplo de datos y resultados está en el fichero `nota_maxmin_vest1.dat`.

Realizad una segunda versión en la cual no solo se obtengan las correspondientes notas, sino el estudiante completo. Para ello, las funciones tendrán que retornar las posiciones de los estudiantes, de forma que en el `main` se pueda gestionar la salida correcta. Para que el resultado sea único hay que introducir un criterio de desempate: en caso de que la nota máxima sea alcanzada por más de un estudiante, las funciones devolverán la posición del que tenga el DNI más pequeño (análogamente con la nota mínima). Un ejemplo de datos y resultados está en el fichero `nota_maxmin_vest2.dat`.

Por último, producid una tercera versión sin suponer que existe al menos un estudiante con nota. Si existe, las funciones han de devolver las posiciones descritas en el apartado enterior; en caso contrario, han de obtener los valores -1 -1 y así la salida correcta podrá determinarse en el `main` con un `if`. Un ejemplo de datos y resultados está en el fichero `nota_maxmin_vest3.dat`. Recordad que solo se puede visitar una vez cada elemento del vector.

2.1.8. Ejercicio: Simplificación de un vector de estudiantes agrupados (*Jutge*)

Consideremos un vector de estudiantes no vacío que puede contener estudiantes repetidos (con el mismo `DNI`, aunque la nota puede variar). Además, supongamos 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 después de ocupar cada posición del vector resultado, no deben visitarse las anteriores.

Notad que el Jutge espera una entrega con un único fichero, por tanto tendréis que programar en él las operaciones de lectura y escritura de vectores. En cualquier caso, algunas de las mejores soluciones requieren modificar ligeramente dichas operaciones.

El programa principal debe tener la estructura

```
declaraciones
llamada a leer_vector
llamada a la funcion de simplificar el vector
llamada a escribir_vector
```

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
```

Un ejemplo de datos y resultados está en los ficheros `purgarvect.dat` y `purgarvect.sal`. 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 más de dos notas y su nota máxima es la primera
- Un estudiante aparece con más de dos notas y su nota máxima es la última
- Un vector con un solo elemento

Tomad las medidas necesarias para que al escribir el vector resultante solo aparezcan los valores significativos. Notad que si en la función declaramos el vector resultado con la dimensión del original, es probable que queden posiciones sin ocupar, que no deseamos escribir. Hay soluciones mejores y peores, por ejemplo se debe evitar recorrer los vectores específicamente para contar cuántos estudiantes diferentes hay o para "borrar" los que no interesan.

Programad una segunda versión tal que la simplificación se realice sobre el vector original, sin usar vectores auxilares.

2.1.9. Medias selectivas de M estudiantes y N asignaturas (*Jutge*, dos versiones)

Consideremos una secuencia de estudiantes, que representa las notas de M estudiantes en N asignaturas. Los valores M y N son enteros mayores que cero y se leen al principio del programa. Cada estudiante tiene exactamente una nota de cada asignatura y, para simplificar, supondremos que todas las notas son distintas de NP (es decir, su valor está entre 0 y 10).

Las notas vendrán agrupadas por estudiante: primero están las N notas del primer estudiante (tras su DNI), después las N del segundo (tras su DNI) y así sucesivamente. Consideramos que para cada estudiante, el orden de aparición de las notas es siempre el mismo, de la primera asignatura a la N-sima.

Adicionalmente, se define un subconjunto de las asignaturas, leyendo primero el tamaño del subconjunto (un valor entero S entre 1 y N) y después S identificadores de asignaturas (valores enteros también entre 1 y N).

Deseamos obtener la media de las notas de cada estudiante en el subconjunto de asignaturas especificado (será por tanto una media de S notas). Realizad tres versiones del ejercicio, suponiendo siempre que el subconjunto de asignaturas se lee antes que las notas. Podéis guardarlo en un vector de N booleanos donde la posición `i` indique si la asignatura `i+1` pertenece al subconjunto.

1. Escribid el DNI y la nota media de los estudiantes en el mismo orden en que éstos aparecen en la secuencia. Notad que para esta versión no hace falta guardar todos los estudiantes en un vector o similares, cada estudiante puede tratarse sobre la marcha.

2. Obtened y escribid un vector de M estudiantes en el que cada estudiante aparezca con su nota media en el subconjunto de asignaturas. El vector solicitado ha de estar ordenado decrecientemente por nota. En caso de existir estudiantes con la misma nota, éstos han de aparecer en orden creciente por DNI. Podéis realizar *una* llamada a la operación genérica `sort` con dicho vector, pero para ello ha de programarse adicionalmente una operación que compare dos estudiantes de la manera adecuada al ejercicio.

3. Dado un valor adicional K, con un valor entre 1 y M, obtened y escribid un vector de K estudiantes que contenga los K estudiantes con la media más alta en el subconjunto de asignaturas. No se puede usar ningún otro vector de estudiantes o equivalente. Escribid el vector según los convenios de ordenación de la versión anterior, pero *sin usar* la operación `sort`.

En los ficheros `mediaselecX.dat` y `mediaselecX.sal`, con `X` $= 1,2,3$ tenéis ejemplos de entrada y salida para las tres versiones.

2.2. La clase `Cjt_estudiants`

Consideremos ahora la clase `Cjt_estudiants` también vista en clase de teoría. Podéis consultar su especificación en el fichero `Cjt_estudiants.pdf`, así como un ejemplo de uso en el fichero

presentats_conj.cc. Tenéis el fichero Cjt_estudiants.hh en la ruta INCLUSIONS y el fichero Cjt_estudiants.o en la ruta OBJECTES.

En este programa se utilizan dos clases, por lo que el proceso completo para obtener el ejecutable presentats_conj.exe será:

```
p2++ -c presentats_conj.cc -I$INCLUSIONS
p2++ -o presentats_conj.exe presentats_conj.o $OBJECTES/{Estudiant.o,Cjt_estudiants.o}
```

2.2.1. Ejercicio: actualizar un conjunto de estudiantes (*Jutge*)

Consideremos dos conjuntos de estudiantes que representan los resultados de dos actos evaluatorios de un mismo grupo de estudiantes. Programad y probad una operación que actualice el primer conjunto, de forma que cada estudiante se quede con la mejor de sus notas de los dos conjuntos originales. Considerad que cualquier nota, incluso cero, es mejor que no tener nota. El segundo conjunto no debe modificarse.

El programa principal debe tener la estructura

1. Declaraciones
2. Leer el primer conjunto mediante llegir (de la clase Cjt_estudiants)
3. Leer el segundo conjunto mediante llegir (de la clase Cjt_estudiants)
4. Actualizar el primer conjunto mediante la nueva operación
5. Escribir el primer conjunto actualizado mediante escriure (de la clase Cjt_estudiants)

Para decidir la mejor estrategia para la actualización (paso 3), fijaos en el coste temporal de las operaciones consultoras (Cjt_estudiants.pdf). En concreto, las operaciones de consulta y modificación del elemento i-ésimo de un conjunto son mucho más eficientes que las correspondientes consultoras y modificadoras por DNI.

En el fichero actual_conj.dat veréis un ejemplo de entrada del programa. La salida esperada se encuentra en el fichero actual_conj.sal.

2.2.2. Ejercicio: gestión integral de un conjunto de estudiantes

Escribid una miniaplicación que lea un conjunto de estudiantes y ofrezca la posibilidad de aplicarle los siguientes tratamientos.

- añadir un nuevo estudiante (opción -1)
- consultar la nota de un estudiante a partir de su DNI (opción -2)
- modificar la nota de un estudiante (opción -3)
- redondear la nota a todos los estudiantes del conjunto (opción -4)
- escribir el conjunto de estudiantes (opción -5)

Organizad el programa principal como un proceso iterativo que, tras leer el conjunto, lea un valor entre -1 y -6 indicativo de la opción que queremos aplicar (la opción -6 será la de terminar el programa), después lea los datos necesarios para ella y proceda a aplicarla.

Notad que las opciones -1, -2, -3 y -5 corresponden a operaciones de la clase, por lo que solo necesitaréis comprobar la precondición de las mismas antes de realizar las correspondientes llamadas. En caso de no cumplirse la precondición no debe realizarse la llamada, simplemente se ha de escribir un mensaje informando de ello. En algunos casos se necesitarán también operaciones de la clase `Estudiant`.

La opción -4 requiere programar una operación nueva en el propio fichero `.cc` que contiene al programa principal. Dicha operación estará basada en operaciones de las clases `Estudiant` y `Cjt_estudiants`.

En el fichero `gest_conj.dat` veréis un ejemplo de entrada del programa. La salida esperada se encuentra en el fichero `gest_conj.sal`. Probad el programa con otros datos, que contengan situaciones no contempladas en dichos ficheros, como por ejemplo que el conjunto esté lleno.