Ejercicio Práctico

Metodología de la Programación II

Dpto. Ciencias de la Computación e Inteligencia Artificial
E.T.S. de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Este documento se ha realizado como apoyo a las prácticas de la asignatura Metodología de la Programación II de las titulaciones de Informática de la Universidad de Granada.

Incluye la descripción de la práctica de modularización propuesta para la asignatura. El documento pdf así como todo el material necesario se encuentran disponibles en la página web de la asignatura.

Antonio Garrido Carrillo (agarrido@decsai.ugr.es)
Javier Martínez Baena (jbaena@decsai.ugr.es)

Índice de contenido

1. Introducción

El objetivo de esta práctica es triple:

1. Practicar con un problema donde es necesaria la modularización. Para desarrollar los programas de esta práctica, el alumno debe crear distintos archivos, compilarlos, y enlazarlos para obtener los ejecutables.
2. Introducirse en los conceptos básicos relacionados con la abstracción, como una motivación e introducción previa al estudio de las clases en C++.
3. Introducirse en el uso de gráficos.

Para ello, nos centraremos en dos programas independientes que manejan matrices. Será necesario incluir código para manejar matrices en memoria dinámica para cualquiera de los dos problemas, reutilizándose para el otro.

El problema que se propone requiere el uso de la abstracción. El programador debe mantener la separación entre la interfaz y la implementación de un módulo, de forma que los cambios en la implementación no afecten a la interfaz, es decir, a la forma en la que cualquier programa haría uso de ese módulo. Así, la solución de la práctica incluye el desarrollo de un tipo de dato abstracto `Matriz`, de forma que podamos realizar cualquier programa que use matrices, accediendo a ese módulo a través de su interfaz.

Para enfatizar las ventajas de la abstracción, el estudiante deberá realizar 4 versiones distintas del tipo de dato matriz. Para ello, se propondrán distintas formas de almacenar los elementos de la matriz, de manera que los programas que usen el tipo de dato serán independientes de estos cambios.

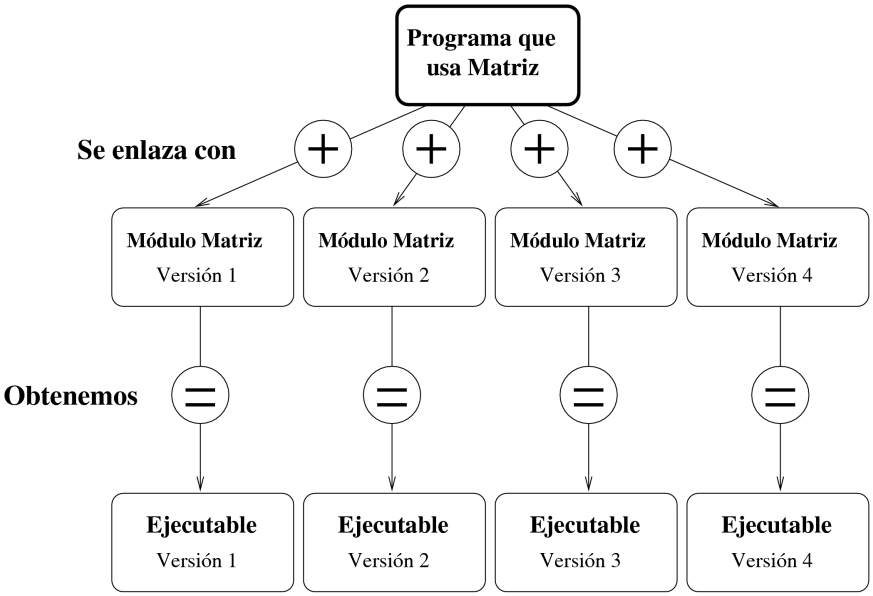

Figura 1: Distintas implementaciones

En la figura 1 se presenta un esquema del resultado que se desea obtener. Podemos observar que:

1. Se presenta el programa que usa el módulo `Matriz`, como la aplicación principal (función `main` y otros módulos adicionales). Aparece una sola vez, ya que sólo es necesario desarrollar una versión.
2. Se presentan 4 versiones del módulo `Matriz`. Cada una de estas versiones considera una forma distinta de almacenar los elementos de una matriz, es decir, distintas representaciones.

3. Se pueden obtener las 4 versiones del programa ejecutable (para cada posible módulo `Matriz`). Si enlazamos el programa con cada una de las versiones de `Matriz`, obtendremos distintos programas ejecutables (con igual funcionalidad) que resuelven nuestro problema.

En los siguientes temas, relacionados con clases en C++, descubrirá que gran parte del esfuerzo para mantener esa abstracción, puede realizarlo el compilador de forma automática.

2. Problemas a resolver

Se desea desarrollar dos aplicaciones para resolver dos problemas independientes:

1. Matriz mágica.
2. Representación gráfica de una matriz.

Como resultado, la práctica del alumno debe permitir generar dos programas ejecutables, que resuelvan cada uno de esos problemas.

2.1. Matriz mágica

Una matriz de enteros es mágica, si es cuadrada (de dimensiones `NxN`), y si:

1. Aparecen todos los números enteros desde 1 a N^2
2. La suma de los elementos de cada fila, cada columna, y las dos diagonales principales dan el mismo valor.

Por ejemplo, la matriz siguiente de dimensiones `5x5` contiene todos los elementos del 1 al 25, y tiene como suma común 65.

$$\begin{pmatrix} 15 & 8 & 1 & 24 & 17 \\ 16 & 14 & 7 & 5 & 23 \\ 22 & 20 & 13 & 6 & 4 \\ 3 & 21 & 19 & 12 & 10 \\ 9 & 2 & 25 & 18 & 11 \end{pmatrix}$$

Desarrolle un programa que lea una matriz, y escriba en la salida estándar si es mágica. Un ejemplo de ejecución de este programa puede ser el siguiente:

```
Introduzca dimensiones: 5 5
Introduzca elementos:
15 8  1  24 17
16 14 7   5 23
22 20 13  6 4
3  21 19 12 10
9  2  25 18 11
La matriz introducida es mágica.
```

2.2. Representación gráfica de una matriz

Una representación muy simple, de la magnitud de cada uno de los elementos de una matriz, se puede realizar si presentamos en una ventana una matriz de colores con intensidades proporcionales al valor de esos elementos.

El algoritmo que se propone consiste en representar cada elemento, con un color rojo en caso de ser negativo, negro si es cero, y verde si es positivo. El algoritmo consiste en:

1. Calcular `min`, el valor mínimo de la matriz.
2. Calcular `max`, el valor máximo de la matriz.
3. Representar el elemento `m` con una intensidad de color y escribir el número en

amarillo. La intensidad correspondiente es:

1. Si `m=0`, el color es el negro.
2. Si `m<0`, el color es rojo con intensidad `(m/min)255`.
3. Si `m>0`, el color es verde con intensidad `(m/max)255`.

Desarrolle un programa que lea una matriz de dimensiones `fxc`, y la represente mediante círculos de colores. Este programa, recibe un parámetro (`T`) por la línea de órdenes, que corresponde al tamaño de cada celda de la matriz, crea una ventana de dimensiones `cTxfT` y dibuja círculos de radio `T/2-2` representando cada uno de los elementos con el color apropiado. Finalmente, escribe los números en amarillo, centrados en cada círculo, con la función simple *graficos::Texto.*

Por ejemplo, la matriz de entrada, de dimensiones `4x5` puede ser la siguiente:

$$\begin{pmatrix} 1 & 8 & -5 & 10 & 3 \\ -3 & -14 & 6 & -5 & 1 \\ -4 & 5 & 2 & 9 & 3 \\ 9 & -6 & 5 & 8 & -10 \end{pmatrix}$$

Si la llamada al programa es:

```
$ circulos 100
Introduzca dimensiones: 4 5
Introduzca elementos:
1   8   -5   10  3
-3  -14   6   -5  1
-4   5    2    9  3
 9  -6    5    8 -10
```

El resultado de representar la matriz anterior se muestra en la figura 2, donde también se puede ver el detalle de los tamaños obtenidos.

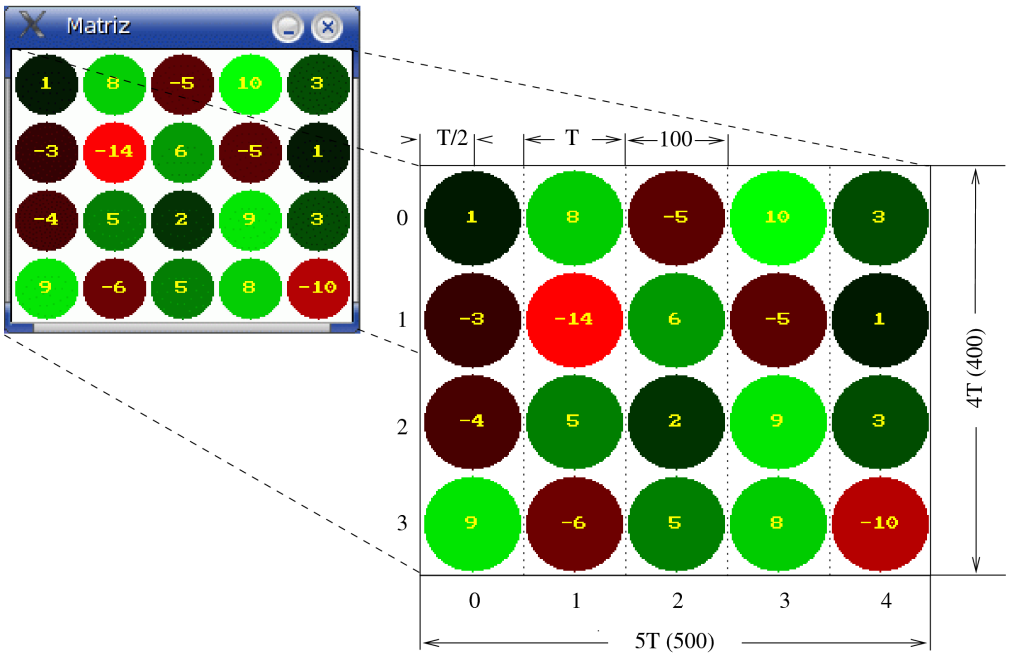

Figura 2: Representación de una matriz

3. Diseño propuesto

Aunque los problemas se pueden resolver de forma independiente, se desea obtener una buena solución modular, de forma que favorezca la reutilización y la abstracción. Dado que las dos aplicaciones están relacionadas con matrices, se propone la creación de un módulo para trabajar con este tipo de dato. Para ello, se crea el tipo `Matriz`, junto con una serie de operaciones para trabajar con ella.

3.1. El módulo matriz

Este tipo de dato se creará en memoria dinámica, para permitir procesar cualquier tamaño. Proponemos la siguiente interfaz:

```
struct Matriz {
    // Implementación....
};

void CrearMatriz (Matriz& m, int f, int c); // Reserva recursos e inicializa m
int FilasMatriz (const Matriz& m);        // Devuelve el número filas de m
int ColumnasMatriz (const Matriz& m);  // Devuelve el número columnas de m
void SetMatriz(Matriz& m, int i, int j, int v);        // Hace m(i,j)=v
int GetMatriz (const Matriz& m, int i, int j);       // Devuelve m(i,j)
void DestruirMatriz (Matriz& m);                   // Libera recursos de m

void LeerMatriz(Matriz& m);
void EscribirMatriz(const Matriz& m);
int Maximo(const Matriz& m);
int Minimo(const Matriz& m);
```

donde la implementación incluye la información del tamaño, junto con los elementos de la matriz, en memoria dinámica. Los detalles concretos de esta implementación no son relevantes para el resto del programa. Para usar una matriz, llamaremos a las funciones que ofrece el tipo `Matriz` (la interfaz), obviando los detalles de implementación, es decir, no accederemos a los miembros de la estructura desde cualquier sitio que no sea una de las funciones propuestas en la interfaz. En la figura 3 se muestra un esquema de una aplicación que usa este módulo.

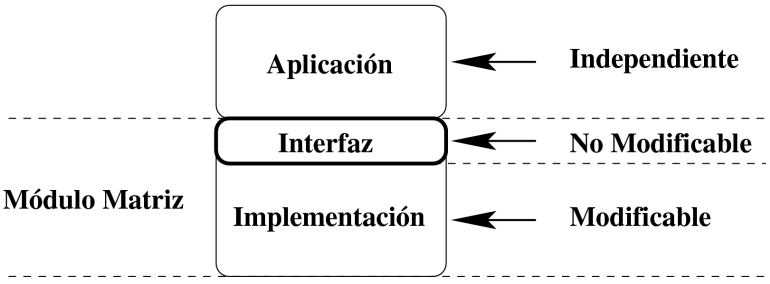

Figura 3: Interfaz común.

Por ejemplo, imagine que la implementación incluye un puntero a un vector donde se encuentran todos los elementos de la matriz (por filas). Podemos crear un ejecutable que use este tipo de representación. Si posteriormente, deseamos modificar la implementación, por ejemplo, con un puntero a un vector de punteros, podemos realizarlo, sin necesidad de modificar el resto de módulos. Observe que los módulos seguirán usando la interfaz, que será la misma, aunque hayamos hecho cambios en la implementación.

Podemos decir, que esa interfaz *encapsula* los detalles de implementación, de forma que lo hace independiente del resto de la aplicación. Note que estos detalles se pueden encapsular

usando solamente las 6 primeras funciones. Es decir, las últimas 4 se pueden programar en base a la interfaz formada por el tipo `Matriz` y las 6 primeras. Esta situación se puede representar como muestra la figura 4, donde distinguimos dos módulos:

1. *Básico*. Incluye el tipo y las funciones básicas para encapsular la implementación.
2. *Adicional*. Incluye funciones adicionales, que usan el módulo anterior.

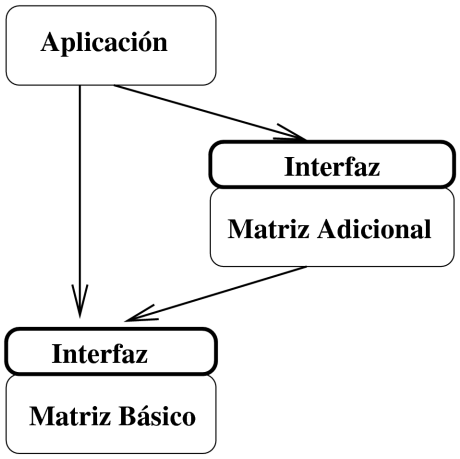

Figura 4: Módulos para matrices

Observe que una modificación en la implementación del tipo `Matriz` resulta bastante simple, ya que sólo afecta a la implementación de las 6 funciones del módulo básico.

Lógicamente, estos módulos se usarán en las dos aplicaciones que hemos propuesto, concretamente:

1. *Mágica*. Esta aplicación se puede realizar añadiendo, a los módulos de gestión de matrices, un módulo que incluya la función `main` para resolver el problema.
2. *Círculos*. Esta aplicación se puede realizar añadiendo, a los módulos de gestión de matrices y de gráficos, un módulo que incluya la función `main` que resuelve el problema.

En la figura 5 se puede ver una representación gráfica de todos los módulos a desarrollar, donde se han enfatizado las interfaces que se relacionan con los módulos principales de cada aplicación. Observe que no es necesario conocer los detalles de las bibliotecas *SDL*, gracias a la capa que establece el módulo de *gráficos* que se facilita.

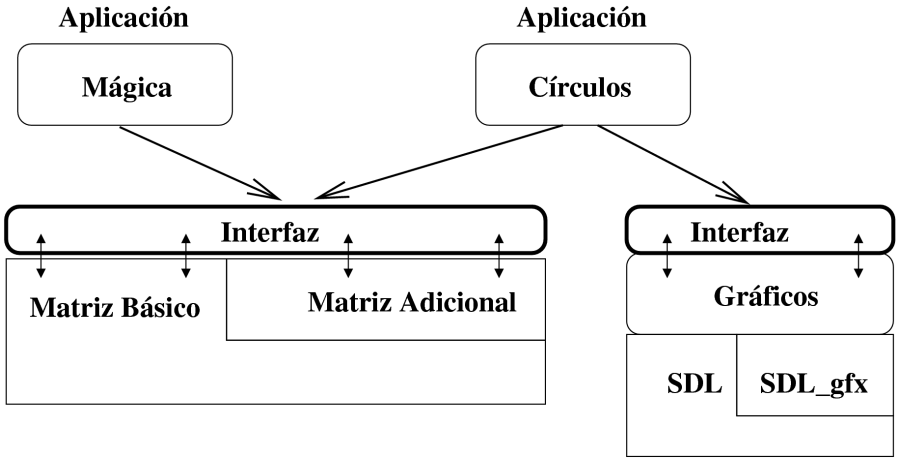

Figura 5: Módulos y dependencias

3.2. Representaciones de matrices

El estudiante debe crear 4 versiones distintas, con diferentes implementaciones, para el tipo de dato `Matriz`. Considere una matriz `m` de tamaño `fxc`. Se proponen las siguientes estructuras de datos en memoria dinámica:

1. Representación basada en un único vector. Las filas de la matriz `m` se almacenan, una tras otra, en un único vector con tamaño `fxc`. En este caso, es necesario un bloque de memoria para todos los datos (una sola operación de reserva de memoria).
2. Representación basada en un vector de punteros a vectores. La matriz `m` se representa con un vector de `f` punteros, cada uno de ellos apuntando a un vector de `c` elementos. En este caso, son necesarios `f+1` bloques de memoria.
3. Representación basada en un vector de punteros a un único vector. La matriz `m` se representa también con un vector de `f` punteros a las filas. Cada puntero apunta a su fila correspondiente, teniendo en cuenta que están almacenadas en un único vector de tamaño `fxc`. En este caso, son necesarios 2 bloques de memoria.
4. Representación basada en celdas enlazadas con punteros a celdas enlazadas. La matriz `m` se representa como un puntero a la primera de `f` celdas enlazadas. Cada celda contiene un puntero a la primera de `c` celdas enlazadas. En este caso, son necesarios `f+fxc` bloques de memoria.

Algunas se presentan con más detalle en [FUN], página 237 y siguientes.

3.3. Selección de implementación

El alumno debe implementar las aplicaciones que se han expuesto en las secciones anteriores. Y por tanto, en principio, si tuviéramos una única representación, podríamos escribir los siguientes archivos:

1. Un archivo *magica.cpp* que resuelve el problema de indicar si una matriz es mágica.
2. Un archivo *circulos.cpp* que resuelve el problema de representar gráficamente una matriz.
3. Dos archivos *matriz_basico.h* y *matriz_basico.cpp*.
4. Dos archivos *matriz_adicional.h* y *matriz_adicional.cpp*, que implementan el módulo de operaciones adicionales.

Se recomienda el desarrollo de estos módulos, así como un archivo *Makefile* para poder trabajar con ellos, antes de abordar el problema de generar automáticamente 4 posibles soluciones modificando la implementación del tipo de dato `Matriz`. Para ello, haga que la generación automática de los ejecutables dependa de la generación de la biblioteca *libmatriz.a*, la cual contendrá el código correspondiente a los archivos "*matriz*cpp*".

Podemos proponer distintas formas para que el paquete software pueda generar el ejecutable seleccionado de forma automática, incluso indicando en la llamada al programa "*make*" la representación deseada, de manera que todo el proceso sea prácticamente transparente para el usuario.

En esta práctica proponemos un método simple que resulta interesante para quien está comenzando a programar. Concretamente, proponemos la siguiente solución para el módulo matrices:

1. Crear un fichero *matriz_basico.h* en el que se incluyan las 4 definiciones para la estructura `Matriz`.
2. Crear un fichero *matriz_basico.cpp* en el que se incluyan las 4 implementaciones (una para cada definición hecha en el .h).
3. Observa que los ficheros *matriz_adicional.h* y *matriz_adicional.cpp* no se verán alterados por estas modificaciones ya que, para su construcción, nos hemos limitado a usar exclusivamente la interfaz del módulo matriz y, en ningún caso, hemos accedido a la parte interna de la representación.

Puesto que tenemos cuatro definiciones distintas para una misma estructura, y eso no es posible en C++, debemos indicarle al compilador cuál de ellas queremos usar en cada compilación que hagamos del módulo. Además, deseamos que esto se pueda hacer de una forma sencilla.

La forma de indicarle a g++ que compile una u otra parte del código escrito en un fichero es mediante las directivas del preprocesador `#if, #ifdef` o `#ifndef`. Por ejemplo, podemos tener el siguiente fichero de cabecera (.h):

```
#define CUAL_COMPILO 1

#if CUAL_COMPILO == 1
       struct A {
         int x,y;
       };
#elif CUAL_COMPILO == 2
       struct A {
         int a, b, c;
         float x,y;
       };
#else    // En cualquier otro caso
       struct A {
         char x, a;
       };
#endif
```

de esta forma, en función del valor que usemos para definir `CUAL_COMPILO`, *g++* compilará una u otra parte de esta estructura condicional, omitiendo el resto de bloques y, por lo tanto, evitando que haya múltiples definiciones de una misma estructura.

Observe que, para nuestro problema, aún cambiando la definición de la estructura `Matriz`, los prototipos de las funciones de la interfaz de *matriz_basico* se mantienen idénticas en los cuatro casos.

De forma análoga, las implementaciones de la interfaz básica del módulo matriz dependerán de la estructura concreta utilizada, por lo que deberemos hacer algo similar para que se compilen sólo aquellas implementaciones que se correspondan con la estructura definida en cada momento.

Esta solución implica que los ficheros *matriz_basico.h* y *matriz_basico.cpp* son relativamente largos, y posiblemente un poco confusos, al contener cuatro implementaciones diferentes para un mismo módulo. Podemos hacer un poco más clara esta modularización si separamos en ficheros diferentes cada cabecera y cada implementación. Es decir, podemos disponer de: *matriz_basico_1.h, matriz_basico_2.h, matriz_basico_3.h, matriz_basico_4.h, matriz_basico_1.cpp, matriz_basico_2.cpp, matriz_basico_3.cpp* y *matriz_basico_4.cpp.* De esta forma, en *matriz_basico.h* -y análogamente en *matriz_basico.cpp*- nos limitaremos a hacer un `#include` del módulo concreto que corresponda, mediante una estructura condicional similar a la del ejemplo anterior.

Para comprobar que se ha realizado todo el proceso correctamente, se propone modificar la definición de la función *CrearMatriz* de forma que la primera acción que lleve a cabo sea escribir en la salida estándar una línea que indique la implementación a la que pertenece. Además, esta línea se incluirá dentro de una directiva `#ifndef` para que sea escrita sólo en caso de que no se esté depurando (véase [FUN], página 325).

Finalmente, y para comprobar cómo se puede eliminar la impresión de esta última línea, compruebe que se puede realizar fácilmente si modificamos las opciones que se pasan al compilador. Concretamente, podemos añadirle "*-DNDEBUG*", de forma que la compilación se realiza como si hubiéramos definido (con `define`) la constante *NDEBUG*.

4. Práctica a entregar

El árbol de directorios de la práctica a entregar deber ser el siguiente:

```
|-- Makefile                Makefile para la generación del proyecto completo (graficos y tetris)
|-- graficos                Carpeta correspondiente al módulo de gestión gráfica
|   |-- Makefile            Makefile para crear la biblioteca gráfica
|   |-- bin                 Ejecutables de prueba para el módulo gráfico
|   |-- data
|   |   `-- fuentes         Fuentes True Type
|   |-- doc                 Documentación del módulo
|   |   |-- doxys           Ficheros auxiliares para doxygen
|   |-- include             Ficheros de cabecera (ficheros .h)
|   |-- lib                 Carpeta con la biblioteca gráfica (ficheros libXXX.a)
|   |-- obj                 Ficheros objeto (ficheros .o)
|   `-- src                 Código fuente (ficheros.cpp)
|
`-- matriz
    |-- Makefile            Fichero makefile para generación de los ejecutables (magica, circulos)
    |-- bin                 Ejecutables
    |-- include             Ficheros de cabecera (ficheros .h)
    |-- lib                 Bibliotecas (ficheros libXXX.a)
    |-- obj                 Código objeto (ficheros .o)
    `-- src                 Código fuente (ficheros .cpp)
```

Teniendo en cuenta lo siguiente:

1. El subárbol de directorios "`graficos`" debe ser el mismo que se ha descargado desde las páginas de la asignatura. Lógicamente, ejecutar la orden "*make*" estando situados en ese directorio provoca la generación de la biblioteca y programas de prueba que se han proporcionado.
2. El subárbol de directorios "`matriz`" corresponde al software que ha desarrollado en esta práctica. Sólo será necesario incluir el código, sin ninguna documentación. Lógicamente, ejecutar la orden "*make*" estando situados en ese directorio provoca la generación de los dos ejecutables en el directorio *bin,* según la representación determinada por la macro definida en el fichero "*include/matriz.h*".
3. El directorio del nivel superior contiene los dos subdirectorios comentados y un archivo "*Makefile*" . El contenido de este fichero permite escribir "*make*" -estando situados en este directorio de nivel superior- para lanzar órdenes "*make*", primero en el directorio "`graficos`", y luego en el directorio "`matriz`". Para ello, consulte el manual de esta orden, concretamente la opción "*-C*".

Los detalles adicionales para poder realizar la entrega podrá encontrarlos en las páginas de la asignatura.

5. Referencias

- [FUN] "*Fundamentos de programación en C++*". A Garrido. Delta publicaciones, 2006.
- [ABS] "Abstracción y estructuras de datos en C++". A Garrido y J. Fdez-Valdivia. Delta publicaciones, 2006.
- [GRF] "Módulo de gestión gráfica". J. Martínez Baena y A. Garrido. Documento de apoyo a MP2. 2008.