Práctica final: clases y E/S

Antonio Garrido / Javier Martínez Baena

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

Metodología de la Programación

Grado en Ingeniería Informática
Curso 2010/2011

Índice de contenido

1. Introducción

Los objetivos de este guión de prácticas son los siguientes:

1. Practicar con un problema en el que 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. Practicar con el uso de la memoria dinámica. El alumno deberá usar estructuras de datos que se alojan en memoria dinámica.
3. Profundizar en los conceptos relacionados con la abstracción de tipos de datos.
4. Practicar con el uso de clases como herramienta para implementar los tipos de datos donde se requiera encapsular la representación.
5. Usar los tipos que ofrece C++ para el uso de ficheros.

Los requisitos para poder realizar esta práctica son:

1. Saber manejar punteros, memoria dinámica, clases y ficheros.
2. Conocer el diseño de programas en módulos independientes, así como la compilación separada, incluyendo la creación de bibliotecas y de archivos *makefile*.
3. Conocer en qué consisten los formatos de imágenes *PGM* y *PPM* que se han dado en el primer ejercicio práctico de la asignatura ([MP2011a]).

El alumno debe realizar esta práctica una vez que haya estudiado los contenidos sobre clases. La práctica está diseñada para que se lleve a cabo mientras se asimila la última parte de la asignatura (clases y ficheros). Así, el alumno puede empezar a realizarla después de haber asimilado los conceptos básicos sobre clases, incluyendo constructores, destructores y sobrecarga del operador de asignación.

Podrá observar que parte del contenido de esta práctica coincide con las prácticas anteriores. Esta información se ha repetido para que este documento sea autocontenido, y en particular para aquellos alumnos que no hayan realizado los ejercicios anteriores.

1.1. Encapsulamiento

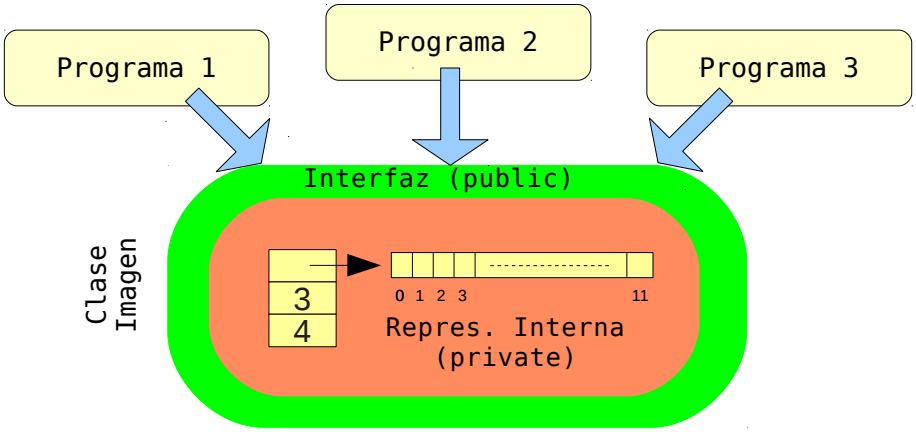
Uno de los objetivos de esta práctica es que el alumno entienda las ventajas del encapsulamiento y cómo las clases en C++ facilitan en gran medida su implementación.

En la práctica anterior ([MP2011b]) se ha presentado el encapsulamiento como una herramienta que facilita el desarrollo y mantenimiento de los programas ya que hacemos que los módulos que usan un tipo sean independientes de los detalles de su representación. En esa práctica, el alumno tenía que ser disciplinado para que sus programas no accedieran a dicha implementación. Para confirmar que todo se realizaba correctamente, proponíamos cambiar la representación para ver que todo el programa seguía siendo válido.

En esta práctica, vamos a encapsular la representación de una imagen, es decir, vamos a crear un módulo para manejar un tipo "*Imagen*". En este módulo encapsulamos la representación con una interfaz, de forma que los programas que lo usen sean independientes de los detalles internos de la representación, ya que sólo necesitarán conocer dicha interfaz.

Para crear este tipo, se usará una clase "Imagen" que garantiza que la representación queda encapsulada. Por tanto, el alumno deberá escoger una representación interna, la que considere más sencilla y eficiente para crear una solución en base a ella. Lógicamente, si alguna vez se deseara un cambio interno de esa representación, tendríamos garantizado que se puede realizar sin problemas, ya que el lenguaje es el que cuidará de que nuestros programas no accedan a esa parte "privada".

La siguiente figura muestra gráficamente esta idea, donde hemos enfatizado la existencia de una clase "Imagen" que contiene dos partes, una pública (la interfaz) y otra privada (la representación interna).



En esta práctica vamos a proponer el desarrollo de varios programas que usan la clase Imagen. Estos programas serán válidos independientemente de la representación interna que seleccionemos. Observe que en la figura anterior se ha mostrado una posible representación interna, aunque el alumno es libre de escoger la que vea conveniente.

1.2. Funciones de E/S de imágenes

El tipo de imagen que vamos a manejar será *PGM* (*Portable Grey Map file format*) , que tiene un esquema de almacenamiento con cabecera seguida de la información. Por tanto, nuestros programas se usarán para procesar imágenes de niveles de gris.

Para simplificar la E/S de imágenes de disco, se facilita un módulo (archivo de cabecera y de definiciones), que contiene el código que se encarga de resolver la lectura y escritura del formato *PGM*. Por tanto, el alumno no necesitará estudiar los detalles de cómo es el formato interno de estos archivos. En lugar de eso, deberá usar las funciones proporcionadas para resolver ese problema. Para más detalles sobre estas funciones, puede consultar [MP2011a].

2. Problemas a resolver

En esta práctica vamos a desarrollar aplicaciones sobre imágenes para resolver varios problemas independientes:

1. Ocultar/Revelar un mensaje.
2. Negativo de una imagen.
3. Desplazamiento de bits.
4. Permutaciones de filas.

Por tanto, la práctica del alumno debe permitir generar los programas ejecutables que se detallan en las secciones siguientes y que resuelven esos problemas.

2.1. Ocultar/Revelar un mensaje

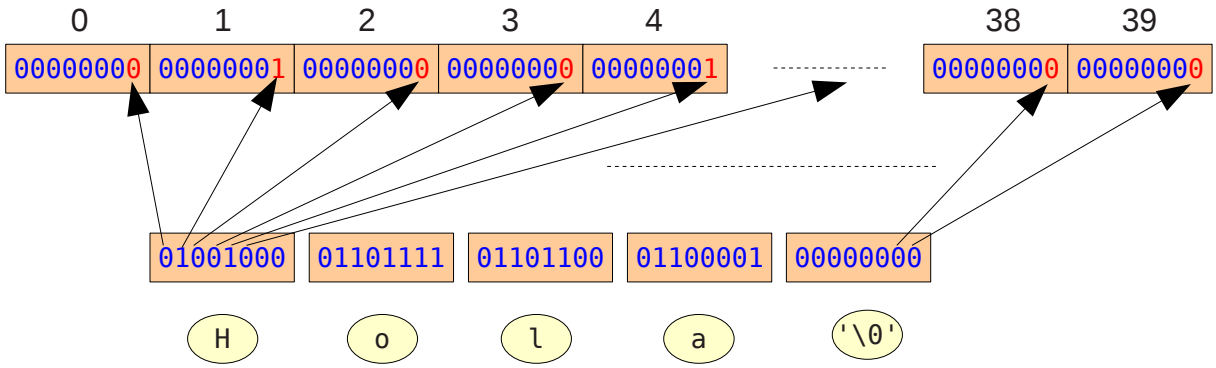
Se van a realizar dos programas para la inserción y extracción de un mensaje “oculto” en una imagen. Para ello, modificaremos el valor de cada píxel para que contenga parte de la información a ocultar. Ahora bien, ¿Cómo almacenamos un mensaje (*cadena-C*) dentro de una imagen?

Tenga en cuenta que los valores que se almacenan en cada píxel corresponden a un valor en el rango [0,255] y que, por tanto, el contenido de una imagen no es más que una secuencia de valores consecutivos en este rango. Si consideramos que el ojo humano no es capaz de detectar cambios muy pequeños en dichos valores, podemos insertar el mensaje deseado modificando ligeramente cada uno de ellos. Concretamente, si cambiamos el valor del bit menos significativo[1], habremos afectado al valor del píxel, como mucho, en una unidad de entre las 255. La imagen la veremos, por tanto, prácticamente igual.

1 *El que representamos a la derecha, y que corresponde a las unidades del número binario.*

Ahora que disponemos del bit menos significativo para cambiarlo como deseemos, podemos usar todos los bits menos significativos de la imagen para codificar el mensaje.

Por otro lado, el mensaje será una *cadena-C*, es decir, una secuencia de valores de tipo *char* que terminan en un cero. En este caso, igualmente, tenemos una secuencia de *bytes* (8 bits) que queremos insertar en la imagen. Dado que podemos modificar los bits menos significativos de la imagen, podemos "repartir" cada carácter del mensaje en 8 píxeles consecutivos. En la siguiente figura mostramos un esquema que refleja esta idea:



Como puede ver, la secuencia de 40 octetos (*bytes*) superior corresponde a los valores almacenados en el vector de "*unsigned char*" que corresponde a la imagen. Podemos suponer, por ejemplo, que la imagen es negra, y que por tanto todos los píxeles tienen un valor de cero.

En la fila inferior, podemos ver un mensaje con 4 caracteres (5 incluyendo el cero final) que corresponde a la secuencia a ocultar. Observe que se han repartido en la secuencia superior, de forma que la imagen ha quedado modificada, aunque visualmente no podremos distinguir la diferencia.

Para realizar la extracción del mensaje tendremos que resolverlo con la operación inversa, es decir, tendremos que consultar cada uno de esos bits menos significativos y colocarlos de forma consecutiva, creando una secuencia de octetos (bytes), hasta que extraigamos un carácter cero.

Por último, es interesante destacar que en el dibujo hemos representado una distribución de *bits* de izquierda a derecha. Es decir, el *bit* más significativo se ha insertado en el primer *byte*, el siguiente en el segundo, hasta el menos significativo que se ha insertado en el octavo. El alumno debe realizar la inserción en este orden y, obviamente, tenerlo en cuenta cuando esté revelando el mensaje codificado.

2.1.1. Ocultar

El programa de ocultación debe insertar un mensaje en una imagen. El programa recibe en la línea de órdenes el nombre de la imagen de entrada y el nombre de la imagen de salida. El mensaje de entrada se carga desde la entrada estándar hasta el fin de entrada. Un ejemplo de ejecución podría ser el siguiente:

```
prompt% ocultar lenna.pgm salida.pgm < mensaje.txt
Ocultando...
prompt%
```

El resultado de esta ejecución deberá ser una nueva imagen en disco, con nombre "*salida.pgm*", que contendrá una imagen similar a "*lenna.pgm*", ya que visualmente será igual, pero ocultará el mensaje que se encuentra en el fichero "*mensaje.txt*".

Lógicamente, esta ejecución corresponde a un caso con éxito, ya que si ocurre algún tipo de error, deberá acabar con un mensaje adecuado. Por ejemplo, en caso de que la imagen indicada no exista, que tenga un formato desconocido, o que el mensaje sea demasiado grande.

2.1.2. Revelar

El programa para revelar un mensaje oculto realizará la operación inversa al anterior, es decir, deberá obtener el mensaje que previamente se haya ocultado con el programa "*ocultar*". Un ejemplo de ejecución podría ser el siguiente:

```
prompt% revelar salida.pgm > resultado.txt
Revelando...
prompt%
```

Observe que hemos usado la misma imagen que se ha obtenido en la ejecución anterior, y el resultado ha sido exitoso al no obtener ningún mensaje de error. Por tanto, en el fichero "*resultado.txt*" tendremos el mensaje original.

De nuevo, tenga en cuenta que si la ejecución encuentra un error, deberá terminar con el mensaje correspondiente.

2.2. Negativo de una imagen

Este programa leerá una imagen *PGM,* y escribirá como salida una nueva imagen *PGM* correspondiente al negativo. Definimos la imagen negativa como una nueva imagen con un tamaño idéntico, en el que el valor de cada píxel se obtiene como el resultado de la siguiente resta:

$$p_{i,j}^{out} = 255 - p_{i,j}^{\text{in}}$$

donde $p_{i,j}$ denota el valor del píxel en la posición *(i,j)* de la imagen. Por tanto, el blanco se convertirá en negro y viceversa.

Un ejemplo de ejecución podría ser el siguiente:

```
prompt% negativo hombro.pgm invertida.pgm
```

donde el parámetro "*hombro.pgm*" es la imagen de entrada (que debe existir en el directorio) y el parámetro "*invertida.pgm*" es una nueva imagen que se generará como resultado del programa.

2.3. Desplazamiento de bits

Este programa leerá una imagen *PGM,* y escribirá como salida una nueva imagen *PGM* correspondiente a la transformación mediante desplazamiento de bits de la imagen de entrada. Definimos la imagen desplazada *n* bits como una nueva imagen con un tamaño idéntico, en el que el valor de cada píxel se obtiene como el resultado de la siguiente operación a nivel de bits:

$$p_{i,j}^{out} = p_{i,j}^{\text{in}} \ll n$$

donde $p_{i,j}$ denota el valor del píxel en la posición *(i,j)* de la imagen, el operador "<<" corresponde al operador de desplazamiento a la izquierda a nivel de bits. Por tanto, en la imagen resultado se perderán los *n* bits más a la izquierda de cada píxel, quedando con un valor que reflejará el valor que había almacenado en los bits menos significativos.

Un ejemplo de ejecución podría ser el siguiente:

```
prompt% desplazar 4 lenna.pgm resultado.pgm
```

donde el parámetro "*lenna.pgm*" es la imagen de entrada (que debe existir en el directorio) y el parámetro "*resultado.pgm*" es una nueva imagen que se generará como resultado del desplazamiento de 4 bits de la imagen de entrada.

2.4. Permutaciones de filas

Vamos a crear un conjunto de aplicaciones que nos permitan transformar una imagen intercambiando la posición de sus filas. En la práctica, para una imagen de *n* filas, podemos representar un intercambio de filas mediante una permutación de los *n* primeros números naturales. Por ejemplo, si tenemos una imagen con 5 filas:

- La permutación 4,3,2,1,0 representa la inversión de las filas, es decir, la imagen reflejada con respecto al eje horizontal.
- La permutación 0,1,2,3,4 representa la identidad, pues cada fila se sitúa en el mismo lugar.
- Etc.

Observe, por tanto, que si modificamos las filas de una imagen con una permutación, en la fila *i* de la nueva imagen se debe colocar la indicada en el valor *i*-ésimo de dicha permutación.

En el material asociado a la práctica, y que podrá descargar desde la página web de la asignatura, encontrará archivos con extensión "*per*", que contienen varios ejemplos de permutación para 512 valores. Tenga en cuenta que si tenemos una imagen con *n* filas, la permutación debe tener *n* valores.

2.4.1. Generación de una permutación

Este programa tiene como objetivo la posibilidad de generar una permutación de manera aleatoria. Dos ejemplos de su ejecución son:

```
prompt% generar 512 b per1_bin.per
prompt% generar 512 t per2_txt.per
```

Observe que tiene tres parámetros:

1. El número de valores de la permutación. Se hará de forma que coincida con el número de filas de la imagen que queremos modificar.
2. Un carácter ('t' o 'b') indicando si queremos el resultado en formato texto o binario (véase sección 3.2.1.).
3. Nombre del archivo donde guardar el resultado.

Como resultado de las dos ejecuciones, deberán existir dos archivos de permutación válidos para imágenes de 512 filas. Estas permutaciones serán aleatorias, y por tanto, podrán generarse con un algoritmo que genere intercambios aleatorios entre números. Por ejemplo, se puede usar un vector inicial con los valores ordenados desde el 0 al 511, y recorrer dicho vector con un índice *i* para, generando un valor aleatorio, *j*, del 0 al 511 (que implica escoger la fila *j* de la imagen), intercambiar el valor que hay en las casillas *i* y *j*.

2.4.2. Aplicar permutación

Este programa se encargará de leer una imagen y una permutación para obtener una nueva imagen con las filas permutadas. Un ejemplo de su ejecución es:

```
prompt% permutacion lenna.pgm intercambios.per codificado.pgm
```

Después de esta ejecución, y si no hay errores, deberá aparecer un nuevo archivo "*codificado.pgm*" en el disco. Este archivo contiene las mismas filas que la imagen de entrada "*lenna.pgm*", pero permutadas según el archivo "*intercambios.per*".

2.4.3. Deshacer permutación

En esta sección no proponemos un nuevo programa, sino una modificación sobre el anterior. El programa "permutacion" propuesto en la sección anterior con 3 parámetros, debe admitir la posibilidad de incluir uno más en la primera posición para poder deshacer las permutaciones. Un ejemplo de ejecución sería:

```
prompt% permutacion -d codificado.pgm intercambios.per decodificado.pgm
```

El resultado es una nueva imagen "*decodificado.pgm*" que contiene el resultado de aplicar la permutación "*intercambios.per*", pero a la inversa. Observe que este programa coincide con la sección anterior porque la diferencia es muy pequeña, ya que los pasos a realizar serán prácticamente los mismos, pero haciendo que la permutación a aplicar se invierta después de ser leída. Si el programa se ha ejecutado tras aplicar la permutación de la sección anterior, el archivo "*decodificado.pgm*" contendrá de nuevo la imagen de "*Lenna*".

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 aplicaciones están relacionadas con imágenes, se propone la creación de un módulo para trabajar con este tipo de dato. Para ello, se creará la clase *Imagen*, junto con una serie de operaciones para trabajar con ella.

Por otro lado, también vamos a trabajar con permutaciones. Hay que tener en cuenta que una permutación es un objeto un poco especial, ya que si tenemos *n* filas, una permutación serán exactamente los números desde *0* a *n-1*, sin repeticiones, uno detrás de otro y en cualquier orden.

Para separar el problema de la gestión de un conjunto de números con estas características, es adecuado realizar una clase *Permutación* que independice mis programas de su representación, y que garantice que es una permutación válida.

3.1. La interfaz de la clase *Imagen*

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

```
class Imagen {
  private:
    // Implementación....

  public:
    int Filas () const;    // Devuelve el número filas de m
    int Columnas () const; // Devuelve el número columnas de m
    void Set (int i, int j, unsigned char v); // Hace img(i,j)=v
    unsigned char Get (int i, int j) const; // Devuelve img(i,j)
    bool LeerImagen(const char file[]); // Carga imagen file en img
    bool EscribirImagen(const char file[]) const; //Salva img en file
};
```

Observe que:

- La parte interna de la clase *Imagen* no se especifica. Los campos que la componen dependen de la representación que queramos usar para la imagen.
- En esta clase será necesario incluir los constructores, destructor y operador de asignación.

Cuando decimos que nuestros programas no van a acceder a la representación, queremos decir que no deben acceder a ningún campo privado que haya en la clase *Imagen*. En lugar de eso, deberán usar la lista de métodos públicos que permiten manejarla.

Para realizar nuevas operaciones relacionadas con imágenes, puede evaluar si realizarlas dentro o fuera de ella, es decir, como funciones miembro o como funciones externas. Tenga en cuenta que si la función se puede realizar sin acceder a la parte privada, podrá implementarse como función externa, independiente de la representación de la imagen.

3.2. Gestión de permutaciones

Para realizar los programas propuestos es necesario gestionar permutaciones, es decir, tenemos que ser capaces de crear permutaciones, consultar el valor de una permutación, salvar una permutación a disco o cargarla desde disco. Para ello, resulta conveniente definir el tipo "*Permutacion*", como una nueva clase que encapsule toda esta dificultad. Esta clase puede contener funciones como las siguientes:

- Crear una nueva permutación desde un número entero que indique el conjunto de valores que la componen e inicializarla con los valores ordenados.
- Modificar la permutación aleatoriamente.
- Leer la permutación desde un fichero de disco.
- Salvar la permutación a un fichero de disco.
- Etc.

Es de especial interés notar que no se debe incluir una función que modifique una posición concreta de la permutación. Por ejemplo, si tenemos la permutación *1,3,0,2,4* de tamaño *5,* y permitimos cambiar cualquier posición de forma individual, por ejemplo, hacer que la posición cero tenga un cuatro, tendríamos una permutación incorrecta. Recordemos que con esta clase tenemos que encapsular la permutación, y sólo permitimos funciones miembro que garanticen que la permutación sigue siendo válida.

3.2.1. Formato de archivos

En esta práctica necesitamos almacenar permutaciones en disco. Para poder realizarlo de una forma más segura, vamos a determinar un formato de almacenamiento de forma que cuando usamos un archivo, tengamos prácticamente garantizado que corresponde a una permutación.

Nuestros programas tendrán que ser capaces de leer dos tipos de ficheros:

1. Fichero de texto. Este formato estará compuesto por:
    - Una cadena "mágica". La cadena se usa para distinguir este archivo de otros tipos. En este caso la cadena está compuesta por los siguientes 8 caracteres: *MP-PER-T*
    - Un separador. Es decir, un carácter "blanco", normalmente un espacio.
    - Un entero, en texto, que indica el número de valores que contiene la permutación.
    - Un separador. Un carácter "blanco", normalmente un salto de línea.
    - Tantos números enteros como sean necesarios para componer la permutación. Estarán almacenados como texto y separados unos de otros por un carácter "blanco", normalmente un espacio o salto de línea.
2. Fichero binario. Este formato estará compuesto por:
    - Una cadena "mágica". La cadena se usa para distinguir este archivo de otros tipos. En este caso la cadena está compuesto por los siguientes 8 caracteres: *MP-PER-B*
    - Un separador. Es decir, un carácter "blanco", normalmente un espacio.
    - Un entero, en texto, que indica el número de valores que contiene la permutación.
    - Un separador. Un carácter "blanco", normalmente un salto de línea.
    - Tantos datos de tipo entero como sean necesarios para componer la permutación. Estarán almacenados en formato binario, sin separación entre ellos.

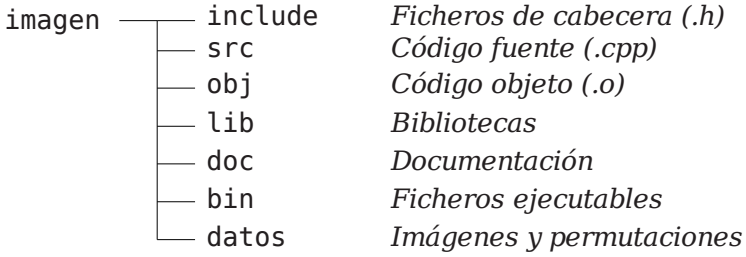
En el material asociado a la práctica, y que puede bajar desde la página web de la asignatura, podrá encontrar archivos con extensión "*per*" (en el directorio datos) con ejemplos de ambos formatos. Para que pueda localizarlos más fácilmente, se ha añadido "*bin*" o "*txt*" al nombre para distinguir a qué formato corresponden. Lógicamente, el nombre del archivo puede ser cualquiera, sin estas letras, o incluso con otra extensión, ya que los programas reconocerán la permutación por la cadena mágica inicial.

4. Práctica a entregar

El alumno deberá empaquetar todos los archivos relacionados en el proyecto en un archivo con nombre "*imagen.tgz*" y entregarlo en la fecha que se publicará en la página web de la asignatura.

Tenga en cuenta que no se incluirán ficheros objeto ni ejecutables. Es recomendable que haga una "limpieza" para eliminar los archivos temporales o que se pueden generar a partir de los fuentes.

Para simplificarlo, el alumno puede ampliar el archivo *Makefile* para que también se incluyan las reglas necesarias que generen los ejecutables correspondientes. Tenga en cuenta que los archivos deben estar distribuidos en directorios*:*



Por consiguiente, lo más sencillo es que comience con la estructura de directorios y archivos que ha descargado desde la página y añada lo necesario para completar el proyecto.

Para realizar la entrega, en primer lugar, realice la limpieza de archivos que no se incluirán en ella, y sitúese en la carpeta superior (en el mismo nivel de la carpeta "*imagen*") para ejecutar:

```
prompt% tar zcvf imagen.tgz imagen
```

tras lo cual, dispondrá de un nuevo archivo *imagen.tgz* que contiene la carpeta *imagen*, así como todas las carpetas y archivos que cuelgan de ella.

5. Referencias

- [GAR06a] Garrido, A. "*Fundamentos de programación en C++*". Delta publicaciones, 2006.
- [GAR06b] Garrido, A. Fdez-Valdivia, J. "*Abstracción y estructuras de datos en C++*". Delta publicaciones, 2006.
- [MP2011a] Garrido, A., Martínez-Baena, J. "*Mensajes e imágenes*". Guión de ejercicio de la asignatura "Metodología de la Programación", curso 2010/2011.
- [MP2011b] Garrido, A., Martínez-Baena, J. "T*ipos de datos abstractos: imágenes*". Guión de ejercicio de la asignatura "Metodología de la Programación", curso 2010/2011.