Práctica final: clases y E/S. Transformación de imágenes

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

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 ([MP2012a]).

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

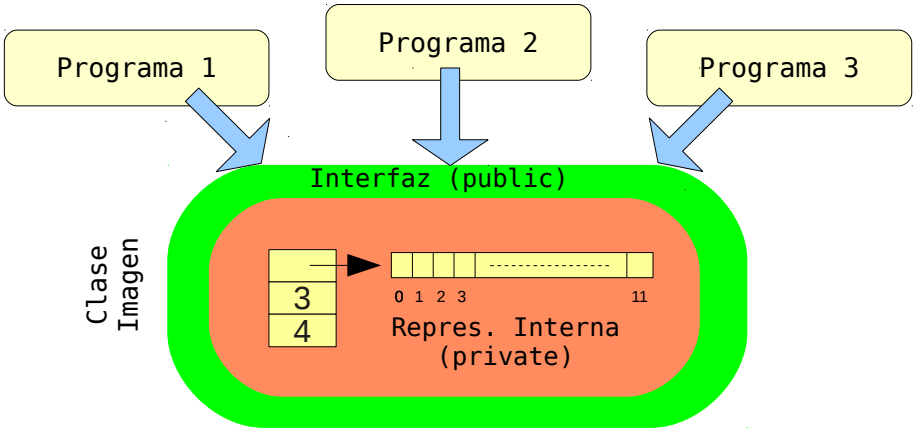
Uno de los objetivos de esta práctica es que el alumno entienda las ventajas del encapsulamiento y cómo las clases en C++ facilitan en gran medida su implementación.

En la práctica anterior ([MP2012b]) 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.

Por otro lado, también se propone la creación de una clase "*Transformación*", para la que vamos a proponer la creación de dos versiones, cambiando la representación interna. En este caso, se podrá comprobar que el cambio afecta solamente a la parte privada de la clase, y que todo el proyecto sigue siendo válido independientemente de la representación utilizada.

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 [MP2012a].

2. Problemas a resolver

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

1. Ocultar/Revelar un mensaje.
2. Realizar transformaciones de los píxeles de la imagen.

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

2.1. Ocultar/Revelar un mensaje

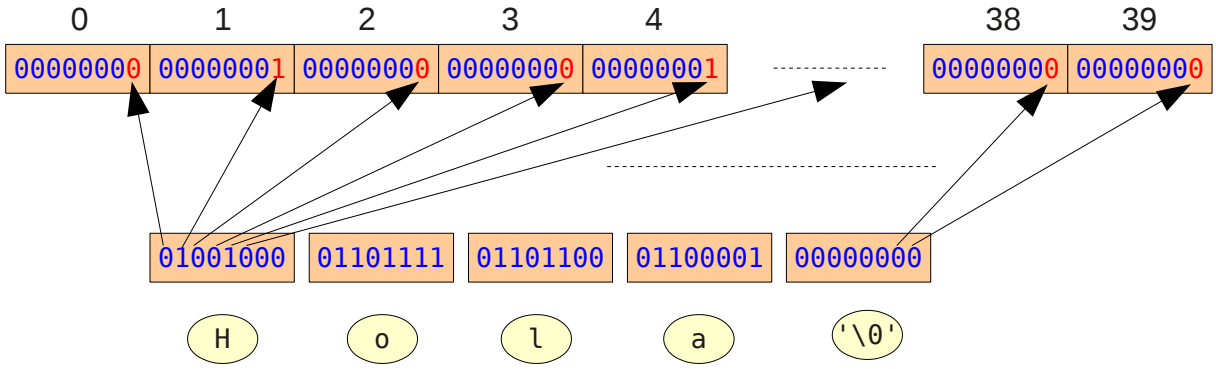
Se van a realizar dos programas para la inserción y extracción de un mensaje "oculto" en una imagen. Para ello, modificaremos el valor de cada píxel para que contenga parte de la información a ocultar. Ahora bien, ¿Cómo almacenamos un mensaje (*cadena-C*) dentro de una imagen?

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.

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*) en la parte superior de la figura 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 de la figura, 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

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

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. Transformaciones de grises.

Una operación muy simple para transformar una imagen consiste en establecer una correspondencia entre los valores de gris de la imagen original y la final. En nuestro caso, en el que tenemos 256 posibles valores, simplemente necesitamos indicar, para cada uno de esos 256 valores, el "valor final" al que se transforma. Podríamos decir que una transformación de grises no es más que una tabla *T*, en la que indicamos que el valor "i-ésimo" se transformará al valor *T(i).*

Por ejemplo, si queremos transformar una imagen de manera que el negro se convierta en blanco, no tenemos más que usar una transformación en la que "*T(0)=255*". Así, cualquier píxel que contenga un valor 0 -negro- en la imagen original, pasará a tener un valor 255 -blanco- en la final.

Observe que la idea es tan simple, que podría usarse un vector de tamaño 256 elementos para manejar transformaciones. Sin embargo, dado que vamos a darle una especial importancia a las transformaciones y queremos añadir operaciones y detalles adicionales a estos "vectores", en esta práctica crearemos un tipo especial "*Transformacion*" para facilitar su uso.

Por ejemplo, una de las posibilidades que se ofrecerá con este tipo será la grabación o lectura de transformaciones en archivos de disco. La idea consiste en que si creamos una transformación, podamos guardarla para poder usarla de nuevo posteriormente, ya sea en la misma imagen o en otras.

En el material asociado a la práctica, y que puede bajar desde la página web de la asignatura, podrá encontrar archivos con extensión "*trf*" (en el directorio datos) con ejemplos. Estos archivos presentan dos tipos de codificación: en texto y en binario. En principio, puede considerar solamente los archivos de texto, ya que podrá ver su contenido con un editor de texto y comprender fácilmente que no son más que una tabla de valores. Más adelante se presentarán los detalles de su formato (sección 3.2.1.).

2.2.1. Transformar

Este programa se encargará de leer una imagen y una transformación para obtener una nueva imagen transformada. Un ejemplo de su ejecución es:

```
prompt% transformar lenna.pgm carboncillo_txt.trf lenna_carbon.pgm
```

Después de esta ejecución, y si no hay errores, deberá aparecer un nuevo archivo "*lenna_carbon.pgm*" en el disco. Este archivo contiene una transformación de la imagen de entrada "*lenna.pgm*", según la transformación almacenada en el archivo "*carboncillo_txt.trf*".

No es difícil intuir que el programa "*transformar*" debe leer una imagen "*Im"* y una transformación "*Tr*", y hacer que el valor que hay en *"Im(i,j)"* pase a valer *"Tr(Im(i,j))"*, para todos los píxeles de la imagen.

2.2.2. Generar transformación

Dado que será necesario disponer de distintos archivos "*trf*" para poder realizar distintas operaciones, es conveniente tener una herramienta que nos facilite la generación. Este programa tiene como objetivo permitir al usuario la generación de archivos de transformación. Ejemplos de su ejecución son:

```
prompt% generar b negativo_bin.trf negativo
prompt% generar t desplazar2_txt.trf desplazar 2
prompt% generar t mitad_txt.trf umbralilzar 127
prompt% generar t brillo10_txt.trf brillo 10
```

Como resultado de las ejecuciones, deberán existir cuatro archivos de transformación, desde el primero que permite hacer un negativo de la imagen hasta el cuarto, que realiza un cambio de brillo en 10 unidades. Observe que tienen diferentes parámetros:

1. Un carácter ('t' o 'b') indicando si queremos el resultado en formato texto o binario (véase sección 3.2.1.).
2. Nombre del archivo donde guardar el resultado.
3. El tipo de transformación que se desea.
4. Si es necesario, otros datos que definen la transformación.

Como puede ver, no es un programa que genere una única transformación, sino que tiene la posibilidad de escoger entre varios tipos.

El programa que implemente debe tener las siguientes posibilidades:

1. Ningún parámetro. En caso de ejecutar "*generar*" sin ningún parámetro, deberá presentar un mensaje de ayuda con los tipos de transformaciones que implementa.
2. Negativo de una imagen. Se solicita con la cadena "*negativo*" como tercer parámetro. Es una transformación en la que el valor de cada píxel se obtiene como el resultado de la siguiente resta:

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

3. Desplazamiento de bits. Se solicita con la cadena "*desplazar*" como tercer parámetro. Además, incluye un cuarto parámetro n que corresponde al número de bits a desplazar. El valor de cada píxel se obtiene como resultado de:

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

4. Umbralización. Se solicita con la cadena "*umbralizar*" como tercer parámetro. Además, incluye un cuarto valor v que corresponde al umbral. El resultado es:

$$p_{i,j}^{out} = \begin{cases} 0 & si \quad p_{i,j}^{\text{in}} \le v \\ 255 & en\,otro\,caso \end{cases}$$

5. Brillo. Se solicita con la cadena "brillo" como tercer parámetro. Además, incluye un cuarto valor d que corresponde al valor a añadir. El resultado es:

$$p_{i,j}^{out} = \begin{cases} 0 & si \quad p_{i,j}^{\text{in}} + d < 0 \\ 255 & si \quad p_{i,j}^{\text{in}} + d > 255 \\ p_{i,j}^{\text{in}} + d & en\,otro\,caso \end{cases}$$

   Observe que el valor de d puede ser positivo o negativo.

Si está interesado en aplicar otros algoritmos de transformación, es libre de añadir los que considere interesantes. Recuerde que si añade alguno nuevo, deberá aparecer en la lista de posibilidades cuando se ejecuta el programa sin parámetros.

2.2.3. Componer

Dadas dos transformaciones *Tr1(v)* y *Tr2(v)*, definimos la transformación composición como:

$$Tr(v)=Tr2(\ Tr1(v)\ )$$

Si queremos aplicar a una imagen varias transformaciones, una detrás de otra, podemos llamar al programa "*transformar*" varias veces. Sin embargo, para simplificar esa operación, podemos crear una transformación que sea composición de las dos, de forma que con una simple llamada podríamos obtener el resultado deseado.

El programa "componer" nos permite obtener una transformación composición a partir de dos existentes. Un ejemplo de su ejecución es:

```
prompt% componer t neg_y_mitad_txt.trf mitad_txt.trf negativo_bin.trf
```

Después de esta ejecución, obtenemos un nuevo archivo transformación "*neg_y_mitad_txt.trf*", codificado en texto, y que corresponde a la composición de los dos siguientes. Así, la transformación corresponde a hacer primero el negativo, y luego umbralizar por la mitad. Observe que los parámetros son:

1. Un carácter ('t' o 'b') indicando si queremos el resultado en formato texto o binario.
2. El nombre del archivo resultado.
3. El nombre del archivo con la segunda transformación.
4. El nombre del archivo con la primera transformación.

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 transformaciones. Hay que tener en cuenta que una transformación es un objeto con unas características particulares. Tiene una serie de correspondencias para todos los posibles valores de niveles de gris, y un conjunto de operaciones que se pueden aplicar en múltiples aplicaciones. Es adecuado realizar una clase *Transformacion* que independice los programas de su representación.

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 de filas de m
    int Columnas () const; // Devuelve el número de 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
    // otros métodos...
};
```

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 transformaciones

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

- Asignar o consultar el valor que corresponde a un nivel de gris.
- Leer la transformación desde un fichero de disco.
- Salvar la transformación a un fichero de disco.
- Etc.

3.2.1. Formato de archivos

En esta práctica necesitamos almacenar transformaciones 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 transformación.

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-TRF-T*
    - Un separador. Es decir, un carácter "blanco", normalmente un espacio.
    - Un entero, en texto, que indica el número de valores que contiene la transformación. En nuestro caso, como usamos un byte, será de 256.
    - Un separador. Un carácter "blanco", normalmente un salto de línea.
    - Tantos números enteros como sean necesarios para componer la transformación. Corresponden a los valores en que se transforman cada uno de los valores de gris. Estarán almacenados como texto y separados unos de otros por un carácter "blanco", normalmente un espacio o salto de línea. En nuestro caso, serán 256 valores enteros.
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 compuesta por los siguientes 8 caracteres: *MP-TRF-B*
    - Un separador. Es decir, un carácter "blanco", normalmente un espacio.
    - Un entero, en texto, que indica el número de valores que contiene la

transformación. En nuestro caso, como usamos un byte, será de 256.

- Un separador. Un carácter “blanco”, normalmente un salto de línea.
- Tantos datos de tipo entero como sean necesarios para componer la transformación. Estarán almacenados en formato binario, sin separación entre ellos.En nuestro caso, serán 256 valores enteros, y como se almacenan en binario con 4 bytes por cada uno, corresponderá a un bloque de 1024 bytes.

En el material asociado a la práctica, y que puede bajar desde la página web de la asignatura, podrá encontrar archivos con extensión “*trf*” (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 transformación por la cadena mágica inicial.

3.2.2. Representación interna del tipo transformación

Para practicar en el desarrollo de clases y ayudar a comprender mejor la importancia del encapsulamiento, el alumno deber realizar dos implementaciones distintas de la clase “*Transformacion*”. En concreto:

1. Un vector estático (no en memoria dinámica) de tamaño fijo 256. Esta representación es la más obvia y simple, ya que sólo es necesario definir en la parte privada un vector de tamaño fijo 256. Observe que en este caso no es necesario realizar constructor de copia ni destructor, ni sobrecargar el operador de asignación.
2. Un vector en memoria dinámica de tamaño 256. El hecho de que sea de tamaño fijo desaconseja esta representación, pero como el objetivo de la práctica es que el alumno realice dos representaciones, también deberá incluir esta posibilidad. En este caso, no se define un vector de tamaño fijo, sino un puntero que apunta a un vector de 256 objetos. Lógicamente, será necesario realizar constructores, destructor y sobrecargar el operador de asignación para que funcione correctamente.

3.2.3. Inicialización y composición de transformaciones

Parte del problema consiste en crear transformaciones concretas. En este caso, podemos plantear dos posiblidades:

(a) Crear un método para cada una de las operaciones dentro de la clase *Transformacion*.

(b) Crear una función externa -no amiga- para cada una de las operaciones.

Dado que nuestro interés es encapsular la representación de la clase “*Transformacion*” para facilitar su modificación y mantenimiento, es recomendable la segunda de ellas.

Por tanto, debemos incluir métodos que nos permitan modificar y consultar cada uno de los valores que componen la transformación. De esa manera, podemos realizar modificaciones sin necesidad de entrar en la parte privada.

Un ejemplo de función que se puede realizar fuera de la clase es la de composición. En esta función, se debe crear una nueva transformación (modificarla) a partir de otras dos transformaciones (consultándolas). Por ejemplo, podemos implementar una función “*negativo*” que modifica los valores de una transformación con algún tipo de función “Set” de la parte pública de la clase.

Para resolver el problema de la composición, el alumno debe sobrecargar el operador suma, es decir, debe definir la función “operator+”, que toma dos objetos de tipo transformación y devuelve una nueva transformación que corresponde a la composición de las dos primeras. Así, la sentencia:

```
Tr = Tr1 + Tr2;
```

asignará a *Tr* la composición *Tr2(Tr1(v))*. Esta función se implementará como función externa a la clase, sin acceso a la parte privada.

3.3. Archivos y selección de la representación

Para desarrollar los distintos programas podríamos generar los siguientes archivos:

- E/S de imágenes: "*imagenES.h*", "*imagenES.cpp*". Contienen las funciones de E/S para leer y escribir imágenes en disco. Estos dos archivos se pueden descargar de la página web de la asignatura.
- Módulo *Imagen*: "*imagen.h*", "*imagen.cpp*". Este módulo implementa el nuevo tipo *Imagen* y las operaciones asociadas.
- Módulo *Transformacion*: "*transformacion.h*", "*transformacion.cpp*". Este módulo implementa el nuevo tipo *Transformacion* y las operaciones asociadas. Podemos añadir también la función de composición (*operator+*) como función externa a la clase.
- Módulo *Transformaciones*: " *Transformaciones.h*", " *Transformaciones.cpp*". Este módulo implementa funciones para inicializar una transformación (desplazamiento, negativo, etc.).
- Módulo *Procesar*: "*procesar.h*", "*procesar.cpp*". Este módulo implementa los algoritmos con imágenes y transformaciones. Incluye algoritmos para ocultar/revelar un mensaje o para transformar una imagen.
- Programas: "*ocultar.cpp", "revelar.cpp",* "*transformar.cpp*", "generar.*cpp*", y "*componer.cpp*". Estos archivos contendrán las funciones *main* que implementan los programas, con posibles funciones auxiliares si las considera necesarias.

Observe que podemos crear una biblioteca con todos los módulos excepto los que incluyen la función *main*. Es decir, podemos crear una biblioteca "*libimagen.a",* que contendría los archivos "*imagenES.o, imagen.o, transformacion.o, transformaciones.o, procesar.o",* con la que enlazar cualquiera de los ejecutables.

Como queremos desarrollar un programa independiente de la representación del tipo "*Transformacion*", podríamos crear dos versiones de los dos archivos -.h y .cpp- para resolver esta clase y copiar en el proyecto el que vayamos a usar. Sin embargo, como queremos comprobar explícitamente que el cambio de representación resulta sencillo y obtiene un resultado válido, se deberán crear los siguientes archivos:

- Módulo *Transformacion*: "*transformacion.h*", "*transformacion.cpp*". Este módulo es el que se usa para crear la biblioteca final, pero contiene simplemente directivas de precompilador. En el archivo cabecera, se define (con "#define") una macro "*CUAL_COMPILO*" que puede tener el valor 1 o 2. Dependiendo de su valor, se incluye la versión 1 o 2.
- Versión 1: "*transformacion1.h*", "*transformacion1.cpp*". Este módulo es el que se incluye si se selecciona la primera representación.
- Versión 2: "*transformacion2.h*", "*transformacion2.cpp*". Este módulo es el que se incluye si se selecciona la segunda representación.

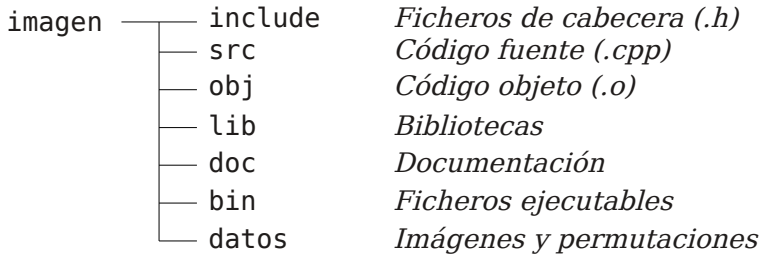
Si quiere ver algún detalle adicional sobre esta forma de seleccionar una representación, puede consultar la práctica [MP2012b] donde se usa el mismo método para seleccionar entre representaciones del tipo *Imagen*. Además, incluye código con detalles de cómo implementarlo.

Finalmente, observe que el módulo "*Transformaciones"* no habría que modificarlo, ya que es un módulo externo y que se quiere implementar independientemente de la representación. Enfatizar este detalle tiene como objetivo que el alumno reflexione sobre la importancia del encapsulamiento. Si una parte del código está correctamente encapsulada, resulta mucho más sencilla su sustitución, modificación, corrección, etc. debido a que el resto del programa no accede a ella. Esta correcta separación está garantizada por el lenguaje, que nos asegura que una parte *private* no puede ser accedida desde fuera de la clase.

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 puedan 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.
- [MP2012a] Garrido, A., Martínez-Baena, J. "*Mensajes e imágenes*". Guión de ejercicio de la asignatura "Metodología de la Programación", curso 2011/2012.
- [MP2012b] Garrido, A., Martínez-Baena, J. "*Tipos de datos abstractos: imágenes*". Guión de ejercicio de la asignatura "Metodología de la Programación", curso 2011/2012.