Tipos de datos abstractos: 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 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. Practicar con el uso de la memoria dinámica. El alumno deberá usar estructuras de datos que se alojan en memoria dinámica.
3. Introducirse en los conceptos relacionados con la abstracción de tipos de datos alojados en memoria dinámica, como una motivación e introducción previa al estudio de las clases en C++.

Los requisitos para poder realizar esta práctica son:

1. Saber manejar punteros y memoria dinámica.
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 punteros y memoria dinámica.

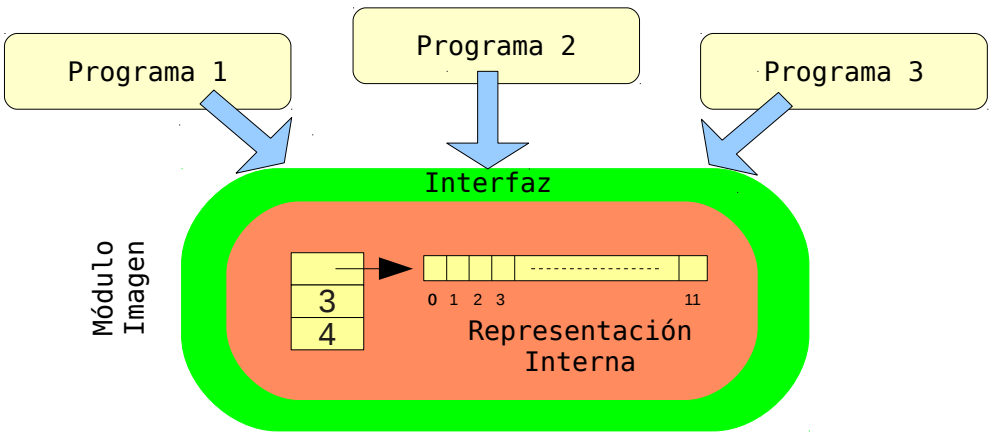
1.1. Encapsulamiento

Uno de los objetivos de esta práctica es que el alumno entienda las ventajas del encapsulamiento.

En temas anteriores ya se ha mostrado que podemos dividir el problema en distintos módulos, de forma que el uso de uno de ellos implica usar la interfaz sin necesidad de conocer los detalles internos de cómo se ha resuelto el módulo.

Un ejemplo lo hemos visto en la práctica de ocultación y revelado de mensajes en una imagen, donde hemos usado un módulo de E/S de imágenes sin necesidad de conocer los detalles de almacenamiento (formatos de imágenes) y los algoritmos necesarios para poder leer o escribir estas imágenes.

En esta práctica, vamos a encapsular la representación de una imagen, es decir, vamos a crear un módulo para manejar un nuevo 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. La siguiente figura muestra gráficamente esta idea:



En esta práctica vamos a proponer el desarrollo de varios programas que usan el módulo imagen. Estos programas serán válidos independientemente de la representación interna que seleccionemos. Para mostrar su validez, se realizarán cambios en la representación interna, obteniendo nuevos ejecutables que, usando otra representación, obtienen los mismos resultados.

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 grises.

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 dos aplicaciones sobre imágenes para resolver dos problemas independientes:

1. Negativo de una imagen.
2. Desplazamiento de bits.

Por tanto, la práctica del alumno debe permitir generar dos programas ejecutables -*negativo* y *desplazar*- que resuelvan cada uno de esos problemas.

2.1. 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.

2.2. 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:

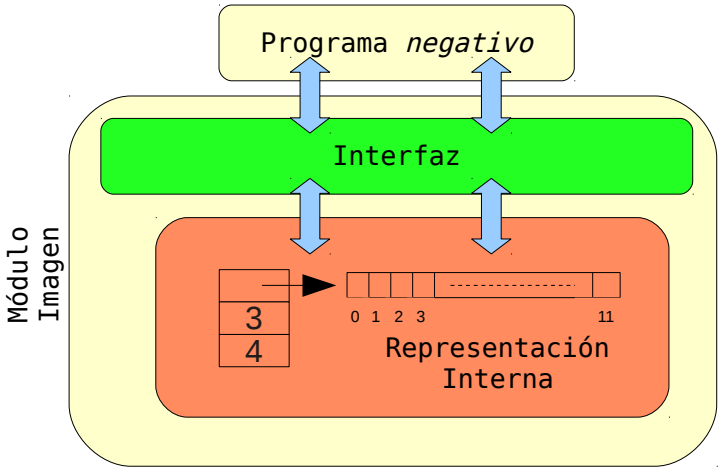
```
prompt% desplazar 4 lenna.pgm resultado.pgm
```

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.

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 imágenes, se propone la creación de un módulo para trabajar con este tipo de dato. Para ello, se crea el tipo *Imagen*, junto con una serie de operaciones para trabajar con él.

Un esquema de los módulos que presenta la abstracción para el programa "*negativo*" se presenta en la figura siguiente:



Podemos observar que el programa hace uso del módulo a través de la interfaz. El programa no necesita acceder a los detalles internos de la representación de la imagen.

3.1. La interfaz del módulo *Imagen*

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

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

void Crear (Imagen& img, int f, int c); // Reserva recursos
int Filas (const Imagen& img);          // Devuelve el número filas de m
int Columnas (const Imagen& img);       // Devuelve el número columnas de m
void Set (Imagen& img, int i, int j, unsigned char v); // Hace img(i,j)=v
unsigned char Get (const Imagen& img, int i, int j);   // Devuelve img(i,j)
void Destruir (Imagen& img);                // Libera recursos de m
bool LeerImagen(const char file[], Imagen& img); // Carga imagen file en img
bool EscribirImagen(const char file[], const Imagen& img); //Salva img en file
```

Observe que:

- La parte interna de la estructura *Imagen* no se especifica. Los campos que la componen dependen de la representación que queramos usar para la imagen.
- En la funciones, pasamos la estructura por referencia cuando la imagen cambia, o como “const &” cuando la imagen no cambia. El uso de “const &” se debe a que queremos pasar la estructura para no modificarla pero queremos evitar tener que hacer la copia correspondiente al paso por valor. La pasamos por referencia indicando que no se modificará.

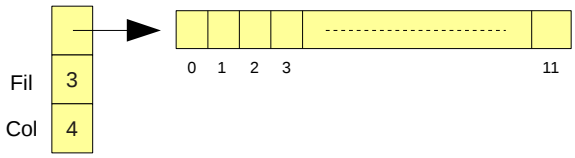
Cuando decimos que nuestros programas no van a acceder a la representación, queremos decir que no deben acceder a ningún campo que haya en la estructura *Imagen*. En lugar de eso, deberán usar la lista de operaciones -funciones- que permiten manejarla. Podemos decir que los campos que declaramos en la estructura son privados al módulo que estamos desarrollando, mientras el conjunto de funciones que proponemos son públicas para los módulos que quieran usar el tipo *Imagen*.

3.2. Representaciones del tipo imagen

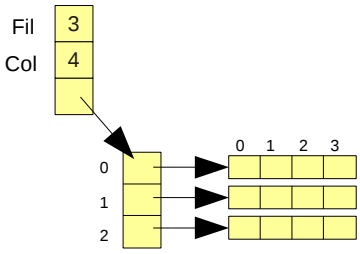
Para poder desarrollar el módulo Imagen será necesario escoger una representación interna para el tipo *Imagen.* Tenga en cuenta que esta representación no será relevante para desarrollar los programas *negativo* o *desplazar*, ya que éstos usan la interfaz del módulo.

El alumno debe realizar 4 versiones del tipo imagen, cada una de ellas con una representación distinta, aunque todas con la misma interfaz. Las representaciones son las siguientes:

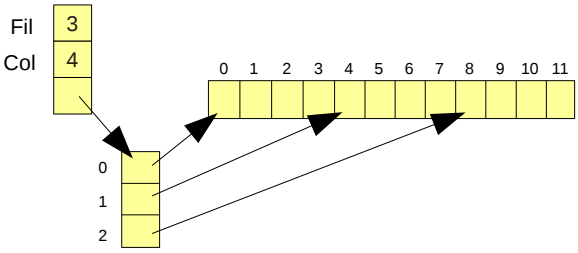
1. Representación en base a dos enteros (filas y columnas) más un vector -en memoria dinámica- con todos los elementos de la imagen consecutivos por filas. En la siguiente figura se muestra un ejemplo de imagen de 3 filas y 4 columnas:



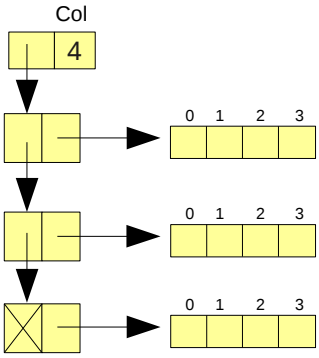
2. Representación en base a dos enteros (filas y columnas) más un vector -en memoria dinámica- de punteros a las filas. Cada puntero apuntará a un vector, también en memoria dinámica, que contiene los elementos de la fila correspondiente. En la siguiente figura se muestra un ejemplo de 3 filas por 4 columnas:



3. Representación en base a dos enteros (filas y columnas) más un vector -en memoria dinámica- de punteros a las filas. El primer puntero apunta a un vector con todos los elementos de la imagen consecutivos por filas. El resto de punteros apunta al comienzo de cada fila. La siguiente figura muestra un ejemplo de 3 filas y 4 columnas:



4. Representación en base a 1 entero (las columnas) más un puntero a una lista de celdas enlazadas (en memoria dinámica) desde las que cuelgan los vectores fila, también en memoria dinámica. La cantidad de filas se obtendrá contando el número de celdas enlazadas.

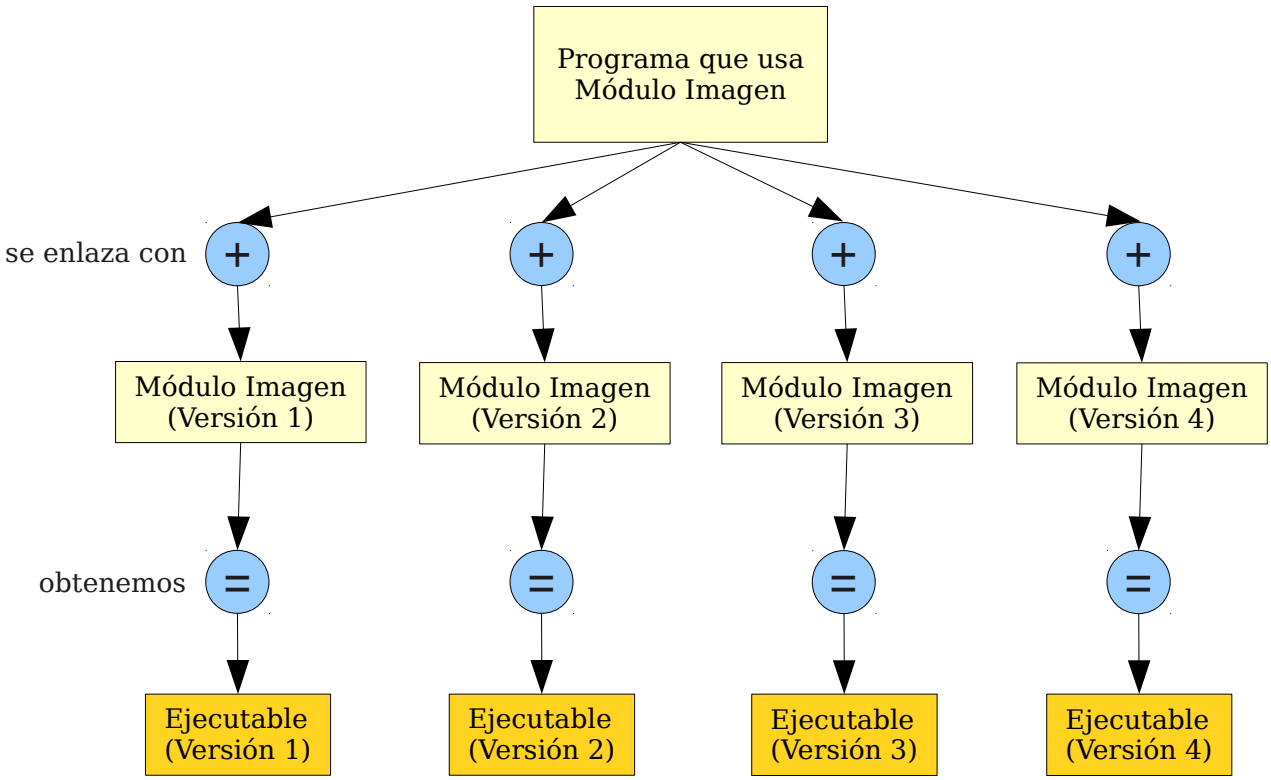


Como puede ver, unas son mejores que otras, ya sea porque ocupan menos espacio, o porque las operaciones del tipo tendrán mejores tiempos de ejecución.

3.3. Comprobando la abstracción

Si realizamos la abstracción correctamente, es decir, implementamos el módulo imagen correctamente, ofreciendo una interfaz independiente de la representación, y el resto de módulos se desarrollan en base a ella, podremos cambiar de representación sin que afecte a nuestros programas.

Obviamente, el cambio de representación deriva en un nuevo ejecutable que puede ser más o menos eficiente, pero que tiene la misma funcionalidad. En la siguiente figura mostramos que podemos obtener 4 ejecutables distintos, que hacen lo mismo, pero que usan las distintas representaciones propuestas.



Es interesante destacar que el programa que usa el módulo imagen no tiene que cambiar en nada. El mismo programa, las mismas funciones, el mismo *main*, será el que se use para enlazarlo con una nueva versión del tipo *Imagen*.

3.4. Archivos y selección de la representación

Para desarrollar los dos programas, *negativo* y *desplazar,* 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.
- Operaciones de imágenes: "*transformar.h*", "*transformar.cpp*". Estos dos archivos implementan las operaciones sobre imágenes. En concreto, incluirán una función para transformar una imagen en su negativo o para desplazar los bits de cada píxel.
- Programas: "negativo*.cpp*", "*desplazar.cpp*". Estos dos archivos contendrán las funciones *main* que implementan los dos programas, con posibles funciones auxiliares si las considera necesarias.

Con estos archivos, podemos crear la biblioteca "*libimagen.a",* que contendría los archivos "*imagenES.o, imagen.o, transformar.o",* con la que enlazar "negativo*.o*" para obtener el primer ejecutable, o "*desplazar.o*" para obtener el segundo.

Como queremos desarrollar un programa independiente de la representación del tipo Imagen, podríamos cambiar los dos archivos "*imagen.h"* e "*imagen.cpp",* usando otra de las representaciones, para obtener de nuevo los ejecutables sin modificar ningún archivo adicional.

Para simplificar la selección de la implementación, vamos a usar las directivas del precompilador de forma que un simple cambio provoque la recompilación del proyecto en base a otra representación. Para poder realizarlo, el alumno debe implementar 4 versiones distintas de los archivos *imagen.h* e *imagen.cpp*:

1. Archivos: "*imagen1.h*", "*imagen1.cpp*". En esta versión se usará la primera representación (dos enteros y un vector).
2. Archivos: "*imagen2.h*", "*imagen2.cpp*". En esta versión se usará la segunda representación (dos enteros, vector de punteros y vectores fila).
3. Archivos: "*imagen3.h*", "*imagen3.cpp*". En esta versión se usará la tercera representación (dos enteros, vector de punteros y un vector con todas las filas).
4. Archivos: "*imagen4.h*", "*imagen4.cpp*". En esta versión se usará la cuarta representación (un entero con las columnas y una lista de celdas enlazadas desde las que cuelgan las distintas filas).

Lógicamente, eso implica que tenemos cuatro versiones de la misma estructura y las mismas funciones. Para poder generar los ejecutables podríamos copiar la versión deseada en los archivos "*imagen.h"* e "*imagen.cpp"* para obtener el esquema de archivos que antes hemos indicado.

Como hemos indicado, en lugar de hacer esa copia, vamos a usar las directivas de compilación condicional. Para ello, recordemos que el precompilador podría incluir código de forma condicional, dependiendo de si se ha hecho una definición. En nuestro caso, hacemos que la versión a compilar dependa del valor de una definición.

En la página web de la asignatura puede descargar el paquete asociado a esta práctica, en el que encontrará los archivos "*imagen.h"* e "*imagen.cpp"* que deberá usar en la compilación. Estudie su contenido para entender cómo vamos a realizar la selección. Lógicamente, esa implementación implica que para resolver el problema deberá implementar los 8 archivos que hemos indicado, con las 4 versiones correspondientes del módulo *Imagen*.

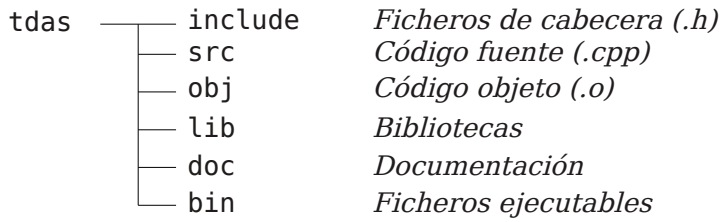
Si lo desea, implemente la primera de ellas ("*imagen1.h*" e "*imagen1.cpp*") y todo el proyecto para confirmar que sus programas funcionan correctamente. Si se han realizado correctamente, podrá implementar la segunda y volver a recompilar los dos programas.

4. Práctica a entregar

El alumno deberá empaquetar todos los archivos relacionados en el proyecto en un archivo con nombre "tdas.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 dos 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 "*tdas*") para ejecutar:

```
prompt% tar zcvf tdas.tgz tdas
```

tras lo cual, dispondrá de un nuevo archivo *tdas.tgz* que contiene la carpeta *tdas*, 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.