UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

Práctica 2: Abstracción. TDA Imagen

(Práctica puntuable)

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

Estructuras de Datos

Grado en Ingeniería Informática
Doble Grado en Ingeniería Informática y Matemáticas
Doble Grado en Ingeniería Informática y ADE

1.- Introducción

1.1 Imágenes de niveles de gris

Una **imagen digital** de niveles de gris puede verse como una matriz bidimensional de puntos (píxeles, en este contexto) en la que cada uno tiene asociado un nivel de luminosidad cuyos valores están en el conjunto {0,1, ..., 255} de forma que el 0 indica mínima luminosidad (negro) y el 255 la máxima luminosidad (blanco). Los restantes valores indican niveles intermedios de luminosidad (grises), siendo más oscuros cuanto menor sea su valor. Con esta representación, cada píxel requiere únicamente un byte (**unsigned char**). En la figura siguiente mostramos un esquema que ilustra el concepto de una imagen con {**alto** filas y {**ancho** columnas. Cada una de las alto x ancho celdas representa un píxel o punto de la imagen y puede guardar un valor del conjunto {0,1, ..., 255}.

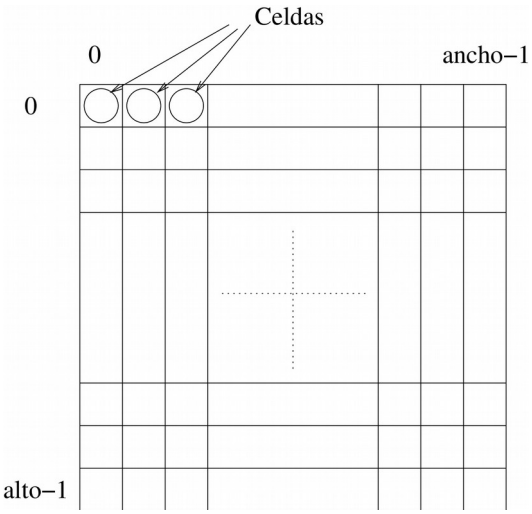

Figura 1.- Esquema de una imagen

Cada casilla de esta matriz representa un punto de la imagen y el valor guardado en ésta indica:
- En imágenes de niveles de gris: su nivel de luminosidad.
- En imágenes en color: su código de color (representación por tabla de color) o la representación del color en el esquema usado (RGB, IHV, etc.).

En esta práctica se trabajará con imágenes de niveles de gris, por lo que cada casilla contiene niveles de luminosidad que se representan con valores del conjunto {0,1, ..., 255} con la convención explicada anteriormente.

A modo de ejemplo, en la figura 2.A mostramos una imagen digital de niveles de gris que tiene 256 filas y 256 columnas. Cada uno de los 256 x 256 = 65.536 puntos contiene un valor entre 0 (visualizado en negro) y 255 (visualizado en blanco). En la figura 2.B mostramos una parte de la imagen anterior (10 filas y 10 columnas) en la que se pueden apreciar, con más detalle, los valores de luminosidad en esa subimagen.

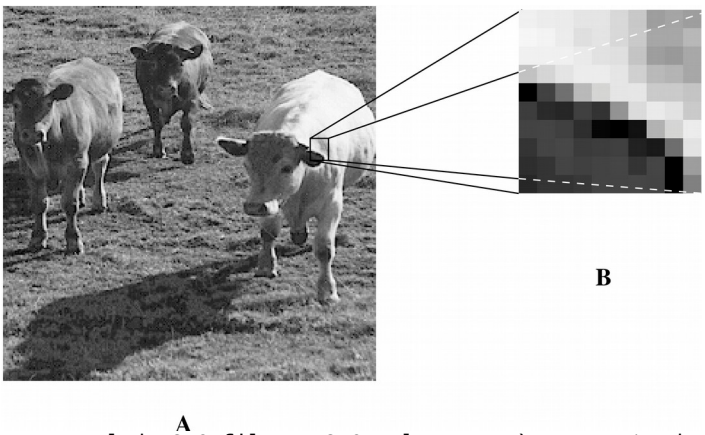

Figura 2.- A. Imagen real de 256 filas y 256 columnas. B) Una parte de esa figura, con 10 filas y 10 columnas

1.2 Imágenes en color

Las imágenes en color se representan de forma diferente ya que un color no puede representarse usando un único valor, sino que se deben incluir tres números. Existen múltiples propuestas sobre el rango de valores y el significado de cada una de esas componentes. Entre los posibles modelos de color, nosotros emplearemos el modelo **RGB**.
Este modelo es muy conocido, ya que se usa en dispositivos como los monitores, donde cada color se representa como la suma de tres componentes: Rojo,Verde y Azul. Podemos considerar distintas alternativas para el rango de posibles valores de cada componente. Por ejemplo,

- Valores reales en el rango [0,1]. El valor 0 indicará que no existe componente de ese color, y el valor 1 que el color contribuye con máxima intensidad.
- Valores enteros en algún rango [0,M]. Con significado similar al caso anterior.

En la práctica, es habitual asignarle el rango de números enteros [0,255], ya que permite representar cada componente con un único byte, y la variedad de posibles colores es suficientemente amplia.

En la figura 2.2 se muestra un ejemplo en el que se crea un color con los valores máximos de rojo y verde, con aportación nula del azul. El resultado es el color (255,255,0), que corresponde al amarillo.

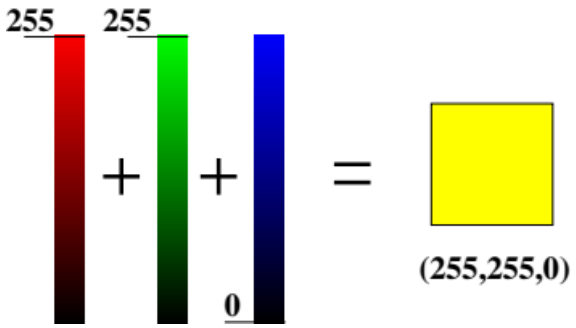

Figura 2.2 Ejemplo del espacio de color RGB.

1.3 Procesamiento de imágenes digitales

El término **Procesamiento de Imágenes Digitales** incluye un vasto conjunto de técnicas cuyo objetivo final es la extracción y representación de información a partir de imágenes digitales. Constituye uno de los campos más atractivos de la Inteligencia Artificial, y en el ámbito científico se conoce con diferentes nombres: Computer Vision, Image Processing, etc., dependiendo del campo y ámbito de aplicación. Una definición informal, y bastante restringida, sería la siguiente: mediante *Procesamiento de Imágenes Digitales* nos referimos a un conjunto de operaciones que modifican la imagen original, dando como resultado una nueva imagen en la que se resaltan ciertos aspectos de la imagen original. Algunas de estas operaciones básicas son:

- **Aumento de contraste**. Tiene como objetivo conseguir una imagen con un contraste mayor (con un rango de niveles de gris más amplio) para resaltar los objetos presentes en la imagen (figura 3.A).

- **Binarización**. Consiste en reducir el número de niveles de gris a dos (blanco y negro) con el objetivo de discriminar ciertos objetos del resto de la imagen (figura 3.B).

- **Detección de bordes**. Consiste en destacar los puntos ``frontera'' entre los objetos presentes en la imagen. Estos bordes se usarán, posteriormente, para segmentar la imagen (figura 3.C).

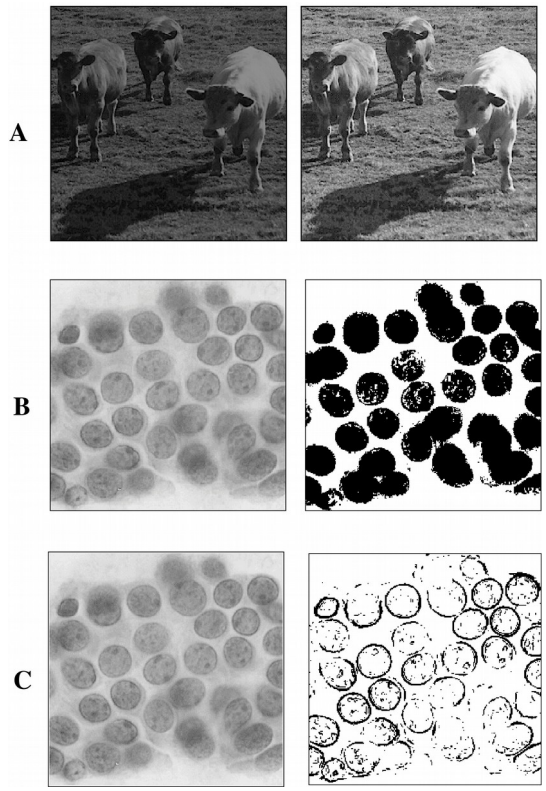

Figura 3. (A) Izquierda: imagen original con 129 niveles de gris (rango: 0 - 128). Derecha: Imagen con más contraste, con 129 niveles de gris (rango: 0 -255). (B) Izquierda: imagen original con 58 niveles de gris (rango: 63 - 239). Derecha: Imagen binaria calculada a partir de la anterior. (C) Izquierda: imagen original con 58 niveles de gris (rango: 63 - 239). Derecha: Imagen con los bordes resaltados.

La lista de operaciones podría ser extensísima, pero con las enumeradas anteriormente basta para dar una idea de las múltiples operaciones que podemos realizar con imágenes digitales.

2.- El TDA imagen

A la hora de implementar programas de manipulación de imágenes digitales debemos plantearnos la conveniencia de utilizar un tipo de dato abstacto (TDA en lo sucesivo) para este propósito. Antes de nada, debemos considerar que previamente a su manipulación se requiere que la imagen resida en memoria y que, una vez que finalice el procesamiento, deberemos liberar la memoria que ocupaba. Con un rápido repaso a las operaciones descritas en la lista anterior nos daremos cuenta que hay dos operaciones básicas necesarias que son comunes: consultar el valor de un punto de la imagen y asignar un valor a un punto de la imagen. Todas las operaciones de procesamiento se pueden hacer en base a estas dos simples operaciones. Realmente no se requiere más para construir un programa de procesamiento de imágenes, aunque es útil disponer de dos operaciones complementarias: consultar el número de filas y consultar el número de columnas de una imagen.

Así, tras este breve análisis podemos plantear ya el TDA imagen. En primer lugar describiremos de forma abstracta el TDA (sección 2.1) a nivel de dominio y operaciones asociadas, para, posteriormente, proponer una representación de los objetos de tipo imagen y una implementación de las operaciones basadas en esa representación.

2.1 Descripción abstracta del TDA imagen

2.1.2 Dominio

Una imagen digital de niveles de gris, **i**, puede verse como una matriz bidimensional de nf filas y nc columnas, en la que en cada elemento, i (f, c), se guarda el nivel de luminosidad de ese punto. El tipo asociado a un objeto de estas características será el tipo **imagen**. Los valores de luminosidad están en el conjunto {0,1, ..., 255} de forma que

el 0 indica mínima luminosidad (y se visualiza con el color negro) y el 255 la máxima luminosidad (y se visualiza con el color blanco). Los restantes valores indican niveles intermedios de luminosidad y se visualizan en distintos tonos de grises, siendo más oscuros cuanto menor sea su valor. El tipo asociado a cada una de las casillas de un objeto de tipo **imagen** será el tipo **byte**.

2.1.2 Operaciones

Dada una imagen digital **i**, definida como

```
Imagen i;
```

las operaciones básicas asociadas son las siguientes:

1. Creación de una imagen.
2. Destrucción de una imagen.
3. Consultar el número de filas de una imagen.
4. Consultar el número de columnas de una imagen.
5. Asignar un valor a un punto de la imagen.
6. Consultar el valor de un punto de la imagen.

que describimos con detalle a continuación:

1. Creación de una imagen.

```
Imagen::Imagen (); // Constructor por defecto
Imagen::Imagen (const Imagen & J); // Constructor de copias
Imagen::Imagen (int filas, int cols);
```

Operadores de tipo constructor.

Propósito: Crear una imagen en memoria con filas filas y cols columnas. Reserva memoria para alojar la imagen.

Precondiciones: filas > 0 y cols > 0.

Postcondiciones:

1. La imagen creada contiene filas filas y cols columnas.
2. La imagen creada contiene valores ``basura'' (no está inicializada).

Ejemplo de uso: Imagen I(10,10); Imagen I; Imagen I(J);

2. Destrucción de una imagen.

```
Imagen::~Imagen();
```

Operador de tipo destructor.

Propósito: Liberar los recursos ocupados por la imagen.
Postcondiciones:

1.Devuelve: nada.
2.La imagen destruida no puede volver a usarse, salvo que se vuelva a realizar sobre ella otra operación `Imagen()`.

3. Consultar el número de filas de una imagen.

```
int Imagen::num_filas () const;
```

Propósito: Calcular el número de filas de la imagen.
Postcondiciones:

1.Devuelve: Número de filas de la imagen.

2.La imagen no se modifica.

Ejemplo de uso: I.num_filas();

4. Consultar el número de columnas de una imagen.

```
int Imagen::num_columnas () const;
```

Propósito: Calcular el número de columnas de la imagen.

Postcondiciones:

1.Devuelve: Número de columnas de la imagen.
2.La imagen no se modifica.

Ejemplo de uso: I.num_columnas();

5. Asignar un valor a un punto de la imagen.

```
void Imagen::asigna_pixel (int fila, int col, unsigned char valor);
```

Propósito: Asignar el valor de luminosidad valor a la casilla (fila, col) de la imagen.
Precondiciones:

1. 0 <= fila < num_filas ()
2. 0 <= col < num_columnas ().
3. 0 <= valor <= 255.

Postcondiciones:

1.Devuelve: nada.
2.La imagen se modifica. Concretamente, se cambia el valor de la casilla (fila, col) de la imagen por el especificado con valor. Los restantes puntos no se modifican.

Ejemplo de uso: I.asigna_pixel(10, 10, 255);

6. Consultar el valor de un punto de la imagen.

```
byte Imagen::valor_pixel (int fila, int col) const;
```

Propósito: Consultar el valor de luminosidad de la casilla (fila, col) de la imagen.

Precondiciones:

1. 0 <= fila < num_filas ()
2. 0 <= col < num_columnas ()

Postcondiciones:

1.Devuelve: El valor de luminosidad de la casilla (fila, col) de la imagen, que está en el conjunto {0,1, ..., 255}.

2.La imagen no se modifica.

Ejemplo de uso: I.valor_pixel(10, 10);

2.1.3 Ejemplos de uso del TDA imagen

Una vez definido de forma abstracta el TDA imagen, mostraremos algunos ejemplos de uso. Obsérvese que en ningún momento hemos hablado de su representación exacta en memoria ni de la implementación de las operaciones. De hecho, este es nuestro propósito, mostrar que puede trabajarse correctamente sobre objetos de tipo imagen independientemente de su implementación concreta.

Ej_Marco

Se trata de enmarcar una imagen plana (todos los puntos tienen el mismo valor). En primer lugar crearemos una imagen plana de nf filas y nc columnas con un valor constante de 255 (blanco) en todos sus puntos. A continuación, ``dibujaremos'' un marco delimitando la imagen. En el siguiente código, nf y nc ya se conocen: pueden haberse leído del teclado o pueden haberse pasado al programa en la línea de órdenes.

```
   int f, c;

   // Construir la imagen de "nf" filas y "nc" columnas

   Imagen I (nf, nc);

   // Crear una imagen plana con valor 255 (blanco)

   for (f=0; f < I.num_filas (); f++)
      for (c=0; c < I.num_columnas (); c++)
         I.asigna_pixel (f, c, 255);

   // Crear el marco que delimita la imagen

   for (f=0; f < I.num_filas (); f++)         // lado izquierdo
      I.asigna_pixel (f, 0, 0);

   for (f=0; f < I.num_filas (); f++)         // lado derecho
      I.asigna_pixel (f, I.num_columnas () - 1, 0);

   for (c=0; c < I.num_columnas (); c++)      // lado superior
      I.asigna_pixel (0, c, 0);

   for (c=0; c < I.num_columnas (); c++)      // lado inferior
      I.asigna_pixel (I.num_filas () - 1, c, 0);
```

Así, podemos crear y rellenar cualquier imagen. Pero no es la mejor manera de proceder, como puede deducirse, ya que este mecanismo resulta muy ``artesanal'': hay que rellenar, pixel a pixel el contenido de la imagen. Para los siguientes ejemplos, supondremos que la imagen se crea y se rellena de alguna forma. Lo usual es que se rellene con datos leídos de un fichero de disco, pero aún no vamos a entrar en ese tema, que lo dejamos para otra sección.

Ahora, si agrupamos en una función las instrucciones que ``dibujan'' el marco, podemos usarla para enmarcar una imagen cualquiera:

```
   // Construir una imagen de "nf" filas y "nc" columnas

   Imagen I (nf, nc);

   // Rellenar la imagen (de alguna forma)
   ......

   enmarca_imagen (I);    // Poner marco

   ......

void enmarca_imagen (Imagen & I)
{
   int f, c,
       nf = I.num_filas(),
       nc = I.num_columnas();

   for (f=0; f < nf; f++)         // lado izquierdo
      I.asigna_pixel (f, 0, 0);
```

```
   for (f=0; f < nf; f++)         // lado derecho
      I.asigna_pixel (f, nc-1, 0);

   for (c=0; c < nc; c++)      // lado superior
      I.asigna_pixel (0, c, 0);

   for (c=0; c < nc; c++)      // lado inferior
      I.asigna_pixel (nf-1, c, 0);
}
```

Puede pensarse que la función enmarca_imagen() podría ser otra operación asociada al TDA imagen. Sin embargo, no es ésta una operación frecuente (más bien muy poco frecuente) por lo que no consideramos conveniente que forme parte del conjunto de operaciones asociadas al TDA imagen. En otro contexto (programas de retoque fotográfico y dibujo) podría ser interesante que fuera parte del conjunto de operaciones básicas. De ser así, se implementaría de otra forma más eficiente, accediendo a la representación de la imagen, en lugar de utilizar las operaciones primitivas.

Ej_binarización

Este otro ejemplo realiza la binarización de una imagen. Esto es, transforma una imagen de niveles de gris a una imagen binaria. Para ello necesita de un valor umbral, **t**, de forma que aquellos puntos con valor de luminosidad menor que el umbral los convierte a cero y los valores que sean mayor o igual los convierte a blanco. Como antes, suponemos que las dimensiones de la imagen y el valor umbral se proporcionan de alguna forma al algoritmo.

```
   int f, c;

   // Construir una imagen de "nf" filas y "nc" columnas

   Imagen I (nf, nc);

   // Rellenar la imagen (de alguna forma)
   ......

  // Binarizacion de la imagen i

   for (f=0; f < I.num_filas (); f++)
      for (c=0; c < I.num_columnas (); c++)
         if (I.valor_pixel (f, c) <= t )
            I.asigna_pixel (f, c, 0);
         else
            I.asigna_pixel (f, c, 255);
```

Como puede observarse, resulta relativamente simple manejar imágenes con el conjunto de operaciones propuesto. A continuación propondremos una representación para los objetos de tipo imagen.

2.2 Representación del tipo imagen

Aunque una imagen se vea o se contemple como una matriz bidimensional, no es ésta la representación más adecuada para los objetos de tipo imagen. La razón es que el compilador debe conocer las dimensiones de la matriz en tiempo de compilación y ésto no responde a nuestros propósitos al ser demasiado restrictivo. Así, debe escogerse una representación que permita reservar memoria para la imagen en tiempo de ejecución. Resulta evidente que la naturaleza bidimensional de la imagen hace que una buena representación sea la de una matriz bidimensional dinámica. Esto nos permite acceder a cada punto de la imagen a través de la notación mediante índices y simplifica la programación de las operaciones.

Entre las dos alternativas de implementación de matrices bidimensionales mediante vectores de punteros, la elección de una u otra dependerá del tipo de procesamiento que pensemos realizar sobre las imágenes y del conjunto de operaciones primitivas que proporcionemos al usuario del TDA.

En nuestro caso, el conjunto de operaciones primitivas es muy simple y reducido, por lo que la primera representación (filas reservadas de forma independiente y vector de punteros a filas) es más que suficiente. Si pensáramos ofrecer operaciones que recorrieran completamente la imagen (p.e. crear una imagen plana o la binarización de una imagen) podríamos pensar seriamente en la segunda alternativa (reserva completa en una única operación y vector de punteros al inicio de cada fila). Así, el contenido de una imagen de filas filas y cols columnas se puede representar como se indica en la figura 4.

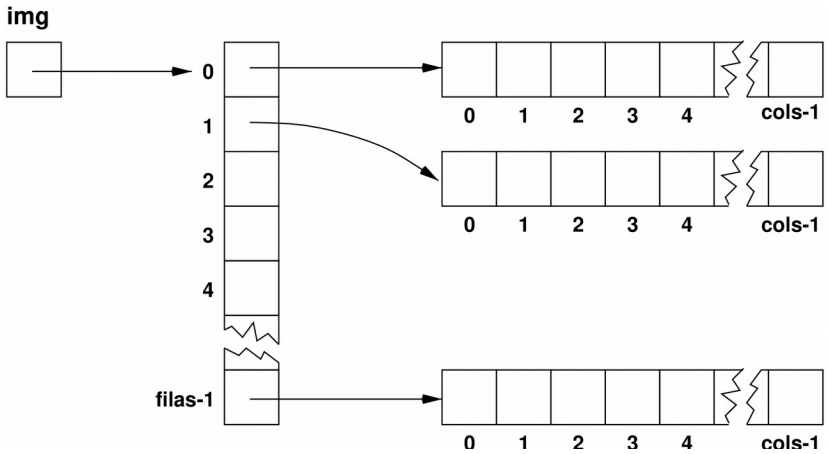

Figura 4. Representación del contenido de una imagen

Sin embargo, esta representación es insuficiente para nuestros propósitos, ya que ofrecemos como operaciones básicas **num_filas() y num_columnas()**, por lo que estos valores deben guardarse, de alguna forma, para cada imagen. Así, el cuerpo básico de una imagen será un registro (**struct**) con tres campos: el número de filas, el número de columnas y un puntero al contenido de la imagen. Finalmente, para simplificar el paso de parámetros a las funciones, podría definirse el tipo imagen como un puntero a un registro del tipo definido anteriormente. En la parte privada de la clase imagen podrían aparecer definiciones del tipo:

```
int filas;     // Numero de filas de la imagen
int cols;      // Numero de columnas de la imagen
unsigned char **img;    // La imagen en si: una matriz dinamica 2D de bytes
```

Con estas definiciones de tipos, la declaración de una variable de la clase imagen llamada **i** sería, p.ej.:

```
Imagen i;
```

que construiría una imagen nula. Para reservar espacio suficiente para albergar una imagen se utilizaría, p.ej., la función constructora Imagen(int filas, int cols). Así, si la imagen **i** va a contener 5000 píxeles dispuestos en 100 filas y 50 columnas, la imagen deberá crearse con la instrucción:

```
Imagen i (100, 50);
```

El número de filas de **i** se obtiene con la función i.num_filas() y el número de columnas con la función i.num_columnas() Finalmente, la función i.valor_pixel(f,c) permite acceder al valor del píxel situado en la fila f y columna c de la imagen **i**.

Podemos agrupar los tipos presentados y los prototipos de las funciones en un fichero de cabecera, al que llamaremos imagen.h. La idea es agrupar las funciones en un fichero llamado imagen.cpp a partir del cual se generará la biblioteca libimag.a. Así, el usuario del TDA dispondrá de la biblioteca (con las funciones compiladas) y del fichero de cabecera asociado (con los tipos descritos anteriormente y los prototipos de las funciones públicas de la biblioteca).

2.2.1. Interface: imagen.h

```
/*****************************************************************************/
// Fichero: imagen.h
// Fichero de cabecera asociado a la biblioteca libimg.a.
// Implementacion del TDA imagen (imagen digital en niveles de gris).
/*****************************************************************************/

#ifndef IMAGEN
#define IMAGEN

 typedef unsigned char byte;  // tipo base de cada pixel

 class Imagen{

 private:

// Definición de los tipos para manejar imagenes digitales. Es solo un ejemplo de lo que
// podría ser esta parte privada.

      int filas;     // Número de filas de la imagen
      int cols;      // Número de columnas de la imagen
      byte **img;    // La imagen en si: una matriz dinamica 2D de bytes

public:

//Es solo un ejemplo de lo que podria ser esta parte pública.
//No se escriben los protocolos, solo la especificación que debería
//hacerse con doxygen y no como está ahora. Además, y aunque no estan especificadas,
//deberían aparecer un constructor por defecto, un constructor de copias y un operador de
//asignación.
/*****************************************************************************/
// Funcion: Imagen(int filas, int cols)
// Tarea:   Crear una imagen en memoria con "filas" filas y "cols" columnas.
//          Reserva memoria para alojar la imagen de "filas" x "cols" pixeles.
// Recibe:  int filas, Número de filas de la imagen.
//          int cols, Número de columnas de la imagen.
// Devuelve: imagen, la imagen creada.
// Comentarios:
//     1. Operación de tipo constructor.
//     2. La imagen creada contiene "filas" filas y "cols" columnas.
//     3. La imagen creada no esta inicializada.
*****************************************************************************/


/*****************************************************************************/
// Funcion: ~Imagen()
// Tarea:   Liberar los recursos ocupados por la imagen.
// Devuelve: void.
// Comentarios:
//     1. Operación de tipo destructor.
//     2. La imagen destruida no puede usarse, salvo que se cree de nuevo.
/*****************************************************************************/


/*****************************************************************************/
// Funcion: int num_filas()  const
// Tarea:   Calcular el numero de filas de la imagen.
// Devuelve: int, Número de filas de la imagen.
// Comentarios: La imagen no se modifica.
/*****************************************************************************/


/*****************************************************************************/
// Funcion: int num_columnas() const
// Tarea:   Calcular el numero de columnas de la imagen "i".
// Devuelve: int, Número de columnas de la imagen.
// Comentarios: La imagen no se modifica.
```

```
/*****************************************************************************/

/*****************************************************************************/
// Funcion: void asigna_pixel(int fila, int col, byte valor);
// Tarea:   Asignar el valor "valor" al pixel ("fila", "col") de la imagen
// Recibe:  int fila, \   fila y columna de la imagen "i"
//          int col,  /   en la que escribir.
//          byte valor, valor a escribir.
// Precondiciones:
//     1. 0 <= "fila" < i.num_filas()
//     2. 0 <= "col" < i.num_columnas()
//     3. 0 <= valor <= 255
// Devuelve: void.
// Postcondiciones:
//     1. "i"("fil","col") == "valor".
//     2. Los restantes píxeles no se modifican.
/*****************************************************************************/

/*****************************************************************************/
// Funcion: byte valor_pixel (int fila, int col)  const
// Tarea:   Consultar el valor de la casilla ("fila", "col") de la imagen
// Recibe:  int fila, \   fila y columna de la imagen "i"
//          int col,  /   a consultar.
// Precondiciones:
//     1. 0 <= "fila" < i.num_filas ()
//     2. 0 <= "col" < i.num_columnas ()
// Devuelve: byte, valor de "i"("fila","col").
// Comentarios: La imagen no se modifica.
/*****************************************************************************/

}

#endif
```

3. Formatos de imágenes

Las imágenes se almacenan en ficheros con un determinado formato. Existen numerosos formatos de imágenes y su descripción detallada puede encontarse en bibliografía específica de Procesamiento de Imágenes o en Internet. Así, para dar versatilidad a nuestra aplicación se requiere un conjunto adicional de funciones para que ``rellenen'' una imagen tomando la información de un fichero o para que guarden una imagen en un fichero. Entre los formatos de imágenes, vamos a trabajar con el formato PGM. Las funciones de lectura y escritura de imágenes van a constituir una biblioteca adicional, que se describe en esta sección.

El formato PGM constituye un ejemplo de los llamados formatos con cabecera. Estos formatos incorporan una cabecera en la que se especifican diversos parámetros acerca de la imagen, como el número de filas y columnas, número de niveles de gris, comentarios, formato de color, parámetros de compresión, etc.

3.1 Descripción del formato PGM

PGM es el acrónimo de **P**ortable **G**ray **M**ap File Format. Como hemos indicado anteriormente, el formato PGM es uno de los formatos que incorporan una cabecera. Un fichero PGM tiene, desde el punto de vista del programador, un formato mixto texto-binario: la cabecera se almacena en formato texto y la imagen en sí en formato binario. Con más detalle, la descripción del formato PGM es la siguiente:

1. Cabecera. La cabecera está en formato texto y consta de:

- Un ``**número mágico**'' para identificar el tipo de fichero. Un fichero PGM que contiene una imagen digital de niveles de gris tiene asignado como identificador los dos caracteres **P5**.

- **Un número indeterminado de comentarios**.

- **Número de columnas** (c).
- **Número de filas** (f).
- **Valor del mayor nivel de gris que puede tener la imagen** (m).

Cada uno de estos datos está terminado por un carácter separador (normalmente un salto de línea).

2. Contenido de la imagen.

Una secuencia binaria de f x c bytes, con valores entre 0 y m. Cada uno de estos valores representa el nivel de gris de un pixel. El primero referencia al píxel de la esquina superior izquierda, el segundo al que está a su derecha, etc.

Algunas aclaraciones respecto a este formato:

- El número de filas, columnas y mayor nivel de gris se especifican en modo texto, esto es, cada dígito viene en forma de carácter.

- Los comentarios son de línea completa y están precedidos por el carácter #+. La longitud máxima es de 70 caracteres. Los programas que manipulan imágenes PGM ignoran estas líneas.

- Aunque el mayor nivel de gris sea m, no tiene porqué haber ningún pixel con este valor. éste es un valor que usan los programas de visualización de imágenes PGM.

En la figura 6 se muestra un ejemplo de cómo se almacena una imagen de 100 filas y 50 columnas en el formato PGM.

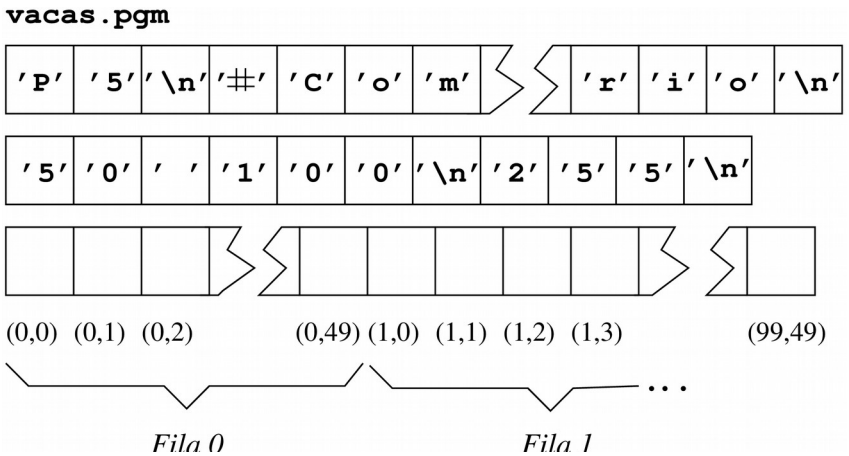

Figura 6. Almacenamiento de una imagen de niveles de gris con 100 filas y 50 columnas en un fichero con formato PGM (vacas.pgm).

3.2 Descripción del formato PPM

PPM es el acrónimo de **P**ortable **P**ix**M**ap File Format. En este caso, para cada celda se almacenan tres valores que corresponden a la cantidad de rojo, verde y azul que se debe mezclar para obtener la imagen deseada (usa el modelo de color RGB). Un fichero PPM tiene el siguiente formato:

1.- **Cabecera:** con un formato prácticamente idéntico al anterior, teniendo en cuenta que:
- La cadena mágica es en este caso ``P6''.
- El valor máximo, m, aparece de la misma forma, indicando que el rango de valores para cada componente es [0,m].

2.- **Contenido de la imagen:** secuencia binaria de f x c tripletas. Cada una de estas tripletas representa el color de un píxel y está representada mediante tres bytes. Esto implica que el rango de posibles valores para cada componente (rojo, verde o azul) será como máximo de 256 (el valor de **m** será como mucho 255). La primera tripleta corresponde al píxel de la esquina superior izquierda, la segunda, al de su derecha, etc. (la última al de la esquina inferior derecha).

Un ejemplo de imagen con este formato es el siguiente:

```
P6
# Creador: ED
# Es una imagen de prueba en color
50 100
255
<secuencia de 50*100*3 bytes>
```

Material disponible.

Para resolver el problema de la entrada y salida de imágenes en el formato PGM, se dispondrá de las siguientes funciones:

```
enum TipoImagen {IMG_DESCONOCIDO, IMG_PGM, IMG_PPM};
TipoImagen LeerTipoImagen (const char *nombre);
unsigned char *LeerImagenPPM (const char *nombre, int& fils, int& cols);
bool EscribirImagenPPM (const char *nombre, const unsigned char *datos,
                        const int fils, const int cols);
unsigned char *LeerImagenPGM (const char *nombre, int& fils, int& cols);
bool EscribirImagenPGM (const char *nombre, const unsigned char *datos,
                        const int fils, const int cols);
```

Y en concreto para las imágenes PGM tenemos:

```
/**
  @brief Leer una imagen en formato PGM.

  @param nombre: Nombre del fichero con la imagen PGM.
  @param filas: Número de filas de la imagen leída.
  @param columnas: Número de columnas de la imagen leída.
  @return Puntero a la zona de memoria con los valores de los píxeles. 0 en caso de
          errores.
Lee una imagen guardada en el fichero indicado por 'nombre' y devuelve el Número de filas
y columnas de la imagen leída (en 'filas' y 'columnas', respectivamente); y un puntero a
un vector con el valor de cada píxel. El vector se aloja en memoria dinámica, (el usuario
es responsable de liberarla) con tamaño exacto de filas x columnas bytes, y almacena el
valor de los píxeles de la imagen de izquierda a derecha y de arriba a abajo.
*/

unsigned char * LeerImagenPGM(const char * nombre, int & filas, int & columnas);

/**
  @brief Guarda una imagen en formato PGM.

  @param v: Vector con los valores de los píxeles de la imagen almacenados por filas.
            Tiene filas x columnas componentes.
  @param nombre: Nombre del fichero con la imagen PGM.
  @param filas: Número de filas de la imagen.
  @param columnas: Número de columnas de la imagen.

  @return true en caso de éxito y false en caso de error.

  Guarda en el fichero indicado por 'nombre' la imagen incluida en 'v' y devuelve un
booleano indicando si ha tenido o no éxito.
 */

bool EscribirImagenPGM(const char * nombre, const unsigned char * v,
                       const int filas, const int columnas);
```

Los protocolos de estas funciones (junto con otros relacionados con la lectura/escritura de imágenes en color) están incluidos en el fichero imagenES.h. Su implementación está incluida en el fichero imagenES.cpp. Se incluye asimismo un ejemplo de uso de estas funciones. El programa negativo.cpp hace uso de las funciones de E/S para el cálculo del negativo de una imagen.

Práctica a realizar

En esta práctica se propone la creación y documentación de la **clase imagen**, que debe incluir (además de las funciones especificadas anteriormente en la sección 2.2.1 de Interface), el **constructor por defecto, el constructor de copias y el operador de asignación.** Una vez construida la clase, se ha de hacer un programa de prueba de la misma que incluirá al menos las siguientes funciones simples de procesamiento de imágenes:

0.- Conversión de RGB a niveles de gris (opcional)

La representación RGB de una imagen incluye información de color y de luminosidad. Es posible, mediante una transformación lineal, extraer la información de luminosidad, descartando la información de color mediante la siguiente fórmula:

$$Igris(i, j) = 0{,}2989 \times R(i, j) + 0{,}587 \times G(i, j) + 0{,}114 \times B(i, j)$$

donde Igris es la imagen de niveles de gris resultante, y R, G y B son las respectivas bandas de la imagen en color.

Los parámetros de la función deberían ser:

<fich_E> nombre del fichero que contiene la imagen en color (en formato PPM).
<fich_S> nombre del fichero que contendrá la conversión a niveles de gris (en formato PGM).

1.- **Umbralizar una imagen usando una escala de grises**

Consiste en generar a partir de una imagen original, otra imagen con el criterio de que si un pixel de la imagen original tiene un nivel de gris p comprendido en un intervalo definido por 2 umbrales T_1 y T_2, se deja con ese nivel de gris p y en otro caso, se pone a blanco (nivel de gris 255). Por tanto, dada la imagen original O, la imagen transformada T se calcula como:

```
T(i,j) =  255        si   O(i,j) <= T_1 ó O(i,j) >= T_2
          O(i,j)     si   T_1 < O(i,j) <  T_2
```

Los parámetros de la función deberían ser:

<fichE> nombre del fichero que contiene la imagen original.
<fichS> nombre del fichero que contedrá el resultado de la transformación.
T_1, T_2 los valores del intervalo de la umbralización (T_1 < T_2).

2.- **Umbralizar una imagen de forma automática.**

Consiste en generar a partir de una imagen original, otra imagen con el criterio de que si un pixel de la imagen original tiene un nivel de gris p menor o igual que un umbral **T**, se deja con ese nivel de gris p (o se pone a 0, como se prefiera) y en otro caso, se pone a blanco (nivel de gris 255). Por tanto, dada la imagen original **O**, la imagen transformada **T** se calcula como:

```
T(i,j) = 255     si  O(i,j) <= T
         O(i,j)  en~otro~caso
```

El umbral T debe determinarse de forma automática siguiendo el siguiente algorítmo iterativo:

(1) Iteración k=0: Calcular el umbral T(k) como el valor de nivel de gris medio de la imagen μ. Con este valor se determinan 2 clases (subimágenes) I_1, I_2 formadas por los píxeles cuya intensidad es menor o igual (resp. mayor) que μ.
(2) Iteración k=1: Para las 2 nuevas clases, calcular sus medias de nivel de gris respectivas μ1 μ2 , y calcular T(k) como

$$T(k) = (\mu 1 + \mu 2)/2$$

(3) Resto de iteraciones: Cada nuevo umbral determina 2 nuevas clases en las que se hace lo mismo que en el paso (2) mientras |T(k+1)-T(k)| >= ε con ε un valor cercano a 0, es decir se recalcula iterativamente el umbral T(k) hasta que su valor se estabilice de una iteración a otra.
(4) El valor final de T(k) es el umbral T que se busca para la umbralización automática.

Los parámetros de la función deberían ser:

<fichE> nombre del fichero que contiene la imagen original.
<fichS> nombre del fichero que contedrá el resultado de la transformación.
T el valor del umbral calculado de forma automática.

3.- Zoom de una porción de la imagen

Consiste en realizar un zoom de una porción de la imagen mediante un simple procedimiento de interpolación consistente en construir a partir de una subimagen N x N, una imagen dedimension (2N-1) x (2N-1), poniendo entre cada 2 filas (resp columnas) de la imagen original otra fila (resp. columna) cuyos valores de gris son la media de los que tiene a laizquierda y a la derecha (resp. arriba y abajo). P.ej. a partir de un trozo 3 x 3 se genera una imagen 5 x 5 de la siguiente forma:

```
10   6  10
 6  10   6
10   4  10
```

Se interpola sobre las columnas:

```
10   8   6   8  10
 6   8  10   8   6
10   7   4   7  10
```

Finalmente se interpola sobre las filas:

```
 10   8   6    8  10
  8   8   8    8   8
  6   8  10    8   6
  8 7.5   7  7.5   8
 10   7   4    7  10
```

donde los valores reales se redondean al entero más próximo por exceso (p.ej. 7.5 pasa a ser 8)

Los parámetros de la función deberían ser:

<fichE> nombre del fichero que contiene la imagen original.
<fichS> nombre del fichero que contedrá el resultado del zoom.
(x_1,y_1) (resp. (x_2,y_2)) coordenadas de la esquina superior izquierda (resp. esquina inferior derecha) del trozo de imagen al que se le hará el zoom. Ha de ser un trozo cuadrado.

4.- Crear un icono a partir de una imagen.

Consiste en crear una imagen de un tamaño muy reducido a partir de una imagen original. El algorítmo de reducción consiste básicamente tomar cada pixel de la salida como la media de los n x n píxeles de la entrada si se quiere hacer una reducción de nx. P.ej si la imagen de entrada es:

```
10   6  10  12
 6  10   6   8
10   4  10  10
12  10   6   8
```

y se hace una reducción 2x quedaría la imagen:

```
 8  9
 9  9
```

donde los valores reales se redondean al entero más próximo por exceso (p.ej., 8.5 pasa a ser 9).

Los parámetros de la función deberían ser:

<fich_orig> nombre del fichero que contiene la imagen original.
<fich_rdo> nombre del fichero donde se guardará el icono.
nf y nc número de filas y columnas del icono resultante.

5.- Aumento de contraste de una imagen mediante una transformación lineal.

Aumentar el contraste en una imagen consiste en generar una imagen de niveles de gris con más contraste que la original. Supongamos que el rango de niveles de gris de la imagen es [a,b], o sea, el mínimo nivel de gris de la imagen es a y el máximo es b. Es evidente que:

$$a <= z <= b$$

donde z es el nivel de gris de un pixel cualquiera de la imagen.

Si queremos que el nuevo rango sea [min, max], la transformación lineal a aplicar, t, a los niveles de gris de la imagen inicial, z, será la siguiente:

$$t(z) = z' = min + [((max - min) / (b - a)) * (z - a)] \quad (E1)$$

Debemos comentar con algún detalle la manera en la que se debe implementar la función de cálculo del contraste. En primer lugar debemos considerar que la función se va a evaluar para cada píxel de la imagen por lo que debemos evitar, en lo posible, realizar cálculos redundantes. En segundo lugar, hay que tener en cuenta que en los cálculos intervienen valores reales por lo que hay que prestar atención a los posibles errores por redondeo.

Sobre el primer punto a tener en cuenta, observar que en la expresión E1 anterior, hay una parte constante e independiente del valor de cada píxel a transformar, el cociente:

$$(max - min)/(b - a)$$

Esta es la razón de que se deba calcular fuera de los ciclos que recorren la imagen. El resultado de este cociente se puede guardar en una variable fija.

Sobre el segundo punto, hay que considerar que el valor z' = t(z) (calculado en la variable valor) es de tipo double y a la hora de asignarlo a la imagen contrastada hay que convertirlo a byte. El valor a guardar debe ser el entero más cercano, por lo que debe redondearse a éste.

Los parámetros de la función deberían ser:

<fichE> nombre del fichero que contiene la imagen fuente.
<fichs> nombre del fichero que contedrá el resultado.
min y max los extremos del nuevo rango de la imagen: los valores de los extremos del intervalo de niveles de gris para la imagen resultante.

6.- Efectos especiales: Simulación de un morphing. (voluntario)

El morphing se usa para cambiar una imagen en otra o para proporcionar una transición suave de una imagen a otra creando la ilusión de una transformación. El término está tomado de la industria de los efectos especiales. Una forma de hacer un morphing simple es establecer un proceso para incrementar o decrementar los valores de los píxeles a partir de una imagen fuente hasta que igualen los valores de los correspondientes pixeles en una imagen destino, lo que ocurrirá en un máximo de 256 iteraciones si son 256 los niveles de gris presentes en las imágenes.

Se propone desarrollar una función que simule un procedimiento sencillo de morphing. Los parámetros de la función deberían ser:

<fich_orig> es el nombre del fichero que contiene la imagen inicial de la que se parte
<fich_rdo> es el nombre del fichero que contiene la imagen final a la que se pretende llegar
<fich_intermedios> son los ficheros intermedios (un máximo de 256) que luego habrán de visualizarse en forma de video (p.ej. con el comando animate de linux) para apreciar la transformación.

7.- Esteganografía: Mensajes ocultos en imágenes (voluntario)

El lenguaje C++ ofrece un conjunto de operadores lógicos a nivel de bit para operar con tipos integrales y enumerados, en particular, con tipos carácter y entero. Los operadores son:
Operadores binarios. La operación se realiza para cada par de bits que ocupan igual posición en ambos operandos.
- O exclusivo (exp1 ^ exp2) bit a bit. Es decir, obtiene 1 cuando uno, y sólo uno de los operandos, vale 1.
- **O** (exp1 | exp2). Obtiene 1 cuando alguno de los dos operandos vale 1.
- **Y** (exp1 & exp2). Obtiene 1 sólo si los dos operandos valen 1.
- **Desplazamiento a la derecha** (exp1 >> exp2). Los bits del primer operando se desplazan a la derecha tantos lugares como indique el segundo operando. Por tanto, algunos bits se perderán por la derecha mientras se insertan nuevos bits cero por la izquierda.
- **Desplazamiento a la izquierda** (exp1 << exp2). Los bits del primer operando se desplazan a la izquierda tantos lugares como indique el segundo operando. Por tanto, algunos bits se perderán por la izquierda mientras se insertan nuevos bits cero por la derecha.
- **Operador unario.** La operación se realiza sobre todos y cada uno de los bits que contiene el operando.
Not (~exp1). Obtiene un nuevo valor con los bits cambiados, es decir, ceros por unos y unos por ceros.

En la siguiente figura se muestran algunos ejemplos con los operadores que hemos indicado.

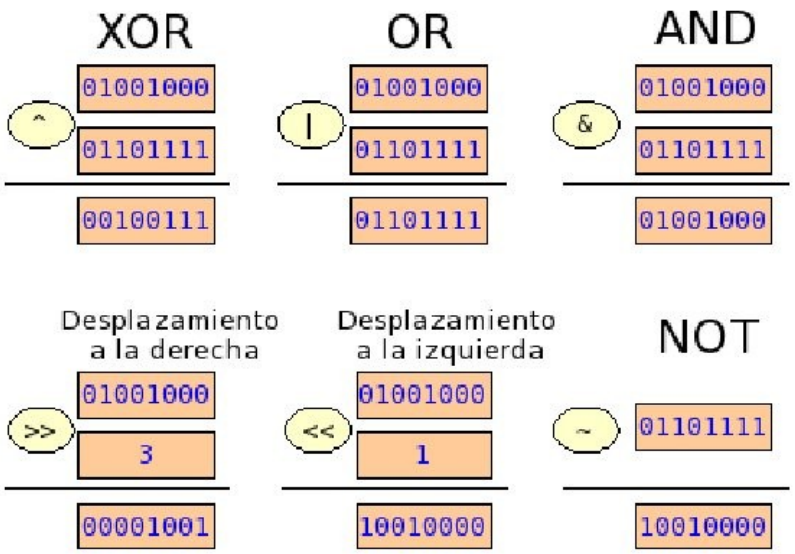

Figura 7.- Operadores lógicos

El ejercicio consiste, en 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? Tened 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 (el que representamos a la derecha, y que corresponde a las unidades del número binario.), 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 (\0). 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:

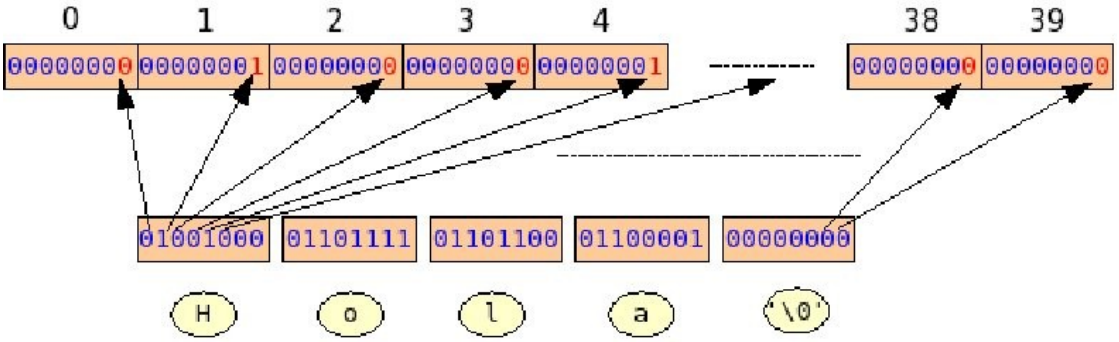

Figura 8.- Esquema de inclusión de un texto

Como se puede ver, la secuencia de 40 octetos (bytes) superior corresponde a los valores almacenados en la estructura de datos 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. Se debe realizar la inserción en este orden y, obviamente, tenerlo en cuenta cuando esté revelando el mensaje codificado.

Para resolver este problema, se tendrán que llevar a cabo una serie de tareas, junto con las condiciones o restricciones que se deberán tener en cuenta.

Módulo codificar

La tarea básica que hay que realizar en el ejercicio consiste en la inserción y extracción de un mensaje en una serie de bytes que componen la imagen. Por tanto, en primer lugar, se propone la creación de un nuevo módulo **<codificar>**, que se encargue de esa tarea y que se use para enlazarse con los programas que se van a desarrollar.

Este módulo contendrá dos funciones:
Función <**Ocultar**>, que recibe como entrada dos parámetros, uno con la imagen (un vector de bytes) y otro con el mensaje a insertar (una cadena-C). Esta función insertará el mensaje en la imagen.
Función <**Revelar**>. Recibe como parámetros la imagen (un vector) y una cadena (un vector de caracteres), que se modificará para contener el mensaje que se va a extraer desde el vector.

Se debe tener en cuenta que se pueden dar situaciones de error y los programas deberán actuar adecuadamente. Ejemplos de posibles situaciones de error:

- La cadena que se intenta codificar es demasiado grande para la imagen dada.
- La imagen codificada no contiene ningún carácter terminador de cadena (carácter '\0').
- La imagen codificada contiene una cadena de tamaño mayor que el parámetro cadena que se le pasa a la función <Revelar>.

Si se considera oportuno, se pueden añadir parámetros adicionales a las dos funciones propuestas para tener en cuenta el tamaño de los vectores y cadenas. De esa forma, las mismas funciones podrán procesar las situaciones de error. Por ejemplo, pueden devolver un valor que indique si ha habido algún error.
También se puede optar por imponer precondiciones a esas funciones, en cuyo caso, se debería comprobar que no hay errores -se cumplen las condiciones- en el lugar de la llamada

Se deben crear los archivos <codificar.h> y <codificar.cpp> para resolver estos dos problemas. Tener en cuenta que puede incluir la devolución de algún valor que indique si se ha conseguido realizar la operación con éxito.

Funciones a implementar:

El objetivo final es crear dos funciones, una para ocultar un mensaje en una imagen y otro para revelarlo.

Ocultar:
La función de ocultación debe insertar un mensaje en una imagen. El programa pide en consola el nombre de la imagen de entrada, el nombre de la imagen de salida, y el mensaje a insertar. Un ejemplo de ejecución podría ser el siguiente:

```
prompt% ocultar
Introduzca la imagen de entrada: lenna.ppm
Introduzca la imagen de salida: salida
Introduzca el mensaje: ¡Hola mundo!
Ocultando...
prompt%
```

El resultado de esta ejecución deberá ser una nueva imagen en disco, con nombre **salida.ppm**, que contendrá una imagen similar a **lenna.ppm**, ya que visualmente será igual, pero ocultará la cadena **¡Hola mundo!**.

Observar que la imagen de salida no incluye la extensión, ya que deberá ser **pgm** si la imagen de entrada está en formato PGM y **ppm** en caso de que sea PPM. Además, el mensaje corresponde a una línea, es decir, deberá leer una cadena de caracteres hasta el final de línea.

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 o tenga un formato desconocido.

Revelar:
La función para revelar un mensaje oculto realizará la operación inversa al anterior, es decir, deberá obtener el mensaje que previamente se haya ocultado con la función <ocultar>. Un ejemplo de ejecución podría ser el siguiente:

```
prompt% revelar
Introduzca la imagen de entrada: salida.ppm
Revelando...
El mensaje obtenido es:
¡Hola mundo!
prompt%
```

Observar que hemos usado la misma imagen que se ha obtenido en la ejecución anterior, y el resultado ha sido exitoso al obtener el mensaje que habíamos ocultado.
De nuevo, tener en cuenta que si la ejecución encuentra un error, deberá terminar con el mensaje correspondiente.

4.- Documentación y entrega

La documentación a entregar en la práctica es la siguiente:

1.- Todo lo necesario para explicar **exhaustivamente** la clase imagen, por lo que deberán detallarse los procesos de abstracción, representación e implementación realizados para su construcción.

2.- **Código**. Listado de los ficheros fuente (.cpp y .h) de los programas y el fichero Makefile.

Una vez hecho lo anterior:

Se enviarán los códigos (así como cualquier otro material que se considere relevante) a través de prado, empaquetando todo de forma estandar en un fichero **tar comprimido** con nombre practica1.tgz **o** en un fichero **zip** con nombre practica1.zip.

Deberán observarse también las siguientes normas generales:

1. La clase imagen, así como todas las funciones que se implementen deberán estar **correctamente documentadas**. Se debería utilizar la herramienta doxygen para ello.

2.- Se debe elegir entre los ejercicios 1 y 2 uno de ellos y entre el 3 y el 4, asímismo uno de los dos solamente

2.- La elaboración de la práctica es **por parejas**. Su valor máximo es de **0.75 puntos si se hace algún ejercicio voluntario (el 6 ó el 7) y 0.5 si no** y su entrega es voluntaria.

3.- La fecha límite de entrega es el **15 de noviembre**. No se admitirá ninguna práctica entregada fuera del plazo establecido.