Guion de prácticas 3

Esteganografía

Marzo de 2015

DECSAI

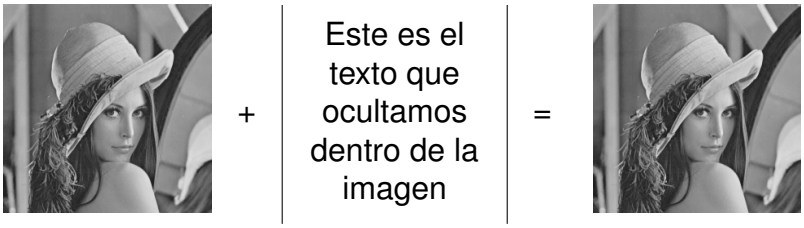
Metodología de la Programación

Curso 2014/2015

Índice

1. Definición del problema

La esteganografía (del griego $\sigma\tau\varepsilon\gamma\alpha\nu o\varsigma$ (steganos): cubierto u oculto, y $\gamma\rho\alpha\varphi o\varsigma$ (graphos): escritura), trata el estudio y aplicación de técnicas que permiten ocultar mensajes u objetos dentro de otros de modo que no se perciba su existencia. Es decir, procura ocultar mensajes dentro de otros objetos y de esta forma establecer un canal encubierto de comunicación, de modo que el propio acto de la comunicación pase inadvertido para observadores que tienen acceso a ese canal[1]. En esta práctica crearemos un sistema de esteganografía para ocultar un mensaje de texto en una imagen.



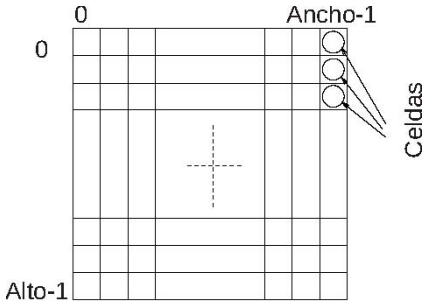
2. Objetivos

El desarrollo de esta práctica pretende servir a los siguientes objetivos:

- manejo de cadenas C y arrays
- practicar el paso de parámetros por referencia
- practicar el paso de parámetros de tipo array y cadenas C
- creación de bibliotecas

3. Imágenes

Desde un punto de vista práctico, una imagen se puede considerar como una matriz bidimensional de celdas, llamadas píxeles, tal como muestra la siguiente figura:



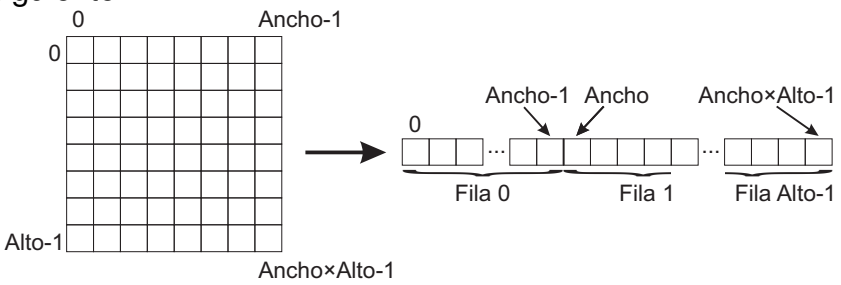
Cada celda de la matriz almacena la información de un píxel. Para imágenes en blanco y negro, cada píxel se suele representar con un byte[2]

[1] Fuente: wikipedia - esteganografía http://es.wikipedia.org/wiki/Esteganograf%C3%ADa

[2] Recuerde que en C++ un "unsigned char" almacena exactamente un byte.

(8 bits). El valor del píxel representa su tonalidad de gris que va desde el negro (0) hasta el blanco (255). Un píxel con valor 128 tendrá un gris intermedio entre blanco y negro. En la siguiente imagen se puede observar el valor de los píxeles para una pequeña porción de una imagen.

Pese a que una imagen se trata habitualmente como una matriz bidimensional de bytes, es usual representarla internamente como un vector en el que las filas se van guardando una tras otra, almacenando consecutivos todos los bytes de la imagen. Así, la posición 0 del vector tendrá el píxel de la esquina superior izquierda, la posición 1 el de su derecha, y así hasta el píxel de la esquina inferior derecha, como se muestra en la figura siguiente:



Así se puede acceder fácilmente a las posiciones de la imagen de forma consecutiva pero, para acceder a cada píxel (x, y) de la imagen es necesario convertir las coordenadas de imagen (x, y) en la coordenada de vector (i). Para ello se aplicará la siguiente fórmula:

$$i = y * columnas + x.$$

Ejercicio 1

Usando esta representación, crear la clase Imagen *según la siguiente especificación:*

```
typedef unsigned char byte;

class Imagen{
private:
private:
   static const int MAXPIXELS = 1000000;   ///< número má
   ximo de píxeles que podemos almacenar
   byte datos[MAXPIXELS]; ///< datos de la imagen
   int nfilas;        ///< número de filas de la imagen
   int ncolumnas;     ///< número de columnsa de la imagen
```

```
public:
   // Construye una imagen vacía (0 filas, 0 columnas)
   Imagen();
   // Construye una imagen negra de tamaño filas x columnas
   Imagen(int filas, int columnas);
   // Crea una imagen negra de tamaño filas x columnas
   void crear(int filas, int columnas);
   // Devuelve el numero de filas de las imagen
   int filas();
   // Devuelve el numero de columnas de la imagen
   int columnas();
   // Asigna el valor v a la posicion (x,y) de la imagen
   void set(int y, int x, byte v);
   // Devuelve el valor de la posicion (x,y) de la imagen
   byte get(int y, int x);
   // Asigna el valor v a la posicion i de la imagen
   considerada como vector
   void setPos(int i, byte v);
   // Devuelve el valor de la posicion i de la imagen
   considerada como vector
   byte getPos(int i);
};
```

Para almacenar las imágenes en el disco duro usaremos el formato *Portable Gray Map* —PGM (http://netpbm.sourceforge.net/doc/pgm.html)— y, para no tener que ocuparnos del formato, se proporcionan los ficheros `pgm.h` y `pgm.cpp` con las funciones básicas de lectura y escritura de imágenes en este formato. El fichero de cabecera `pgm.h` contiene lo siguiente:

```
#ifndef _PGM_H_
#define _PGM_H_
// Tipo de imagen
enum TipoImagen {
  IMG_DESCONOCIDO,   ///< Tipo de imagen desconocido
  IMG_PGM_BINARIO,     ///< Imagen tipo PGM Binario
  IMG_PGM_TEXTO    ///< Imagen tipo PGM Texto
};

// Información sobre la imagen (tipo, filas y columnas)
TipoImagen infoPGM (const char nombre[], int &filas,
  int &columnas);
// Lee de disco una imagen en formato PGM binario
bool leerPGMBinario (const char nombre[], unsigned char
   datos[], int &filas, int &columnas);
// Escribe en disco una imagen en formato PGM binario
bool escribirPGMBinario (const char nombre[], const
   unsigned char datos[], int filas, int columnas);
#endif
```

Además, se incluye documentación en formato doxygen para que sirva de muestra y pueda ser usada como referencia para estas funciones. Ejecute "make documentacion" en el paquete que se le ha entregado para obtener la salida de esa documentación en formato HTML (use un navegador para consultarla).

Ejercicio 2 *Ampliar la clase* Imagen *antes creada con estos dos métodos:*

```
// Carga una imagen desde el fichero
bool leerImagen(const char nombreFichero[]);
// Guarda la imagen en fichero
bool escribirImagen(const char nombreFichero[]);
```

Tanto la lectura como la escritura tratará sólo con imágenes PGM binario. Para leer, el alumno tiene que asegurarse de que la imagen es de tipo `IMG_PGM_BINARIO` *(usando la función* `infoPGM()`*) y que su tamaño es inferior a* `MAXPIXELS` *antes de leer la imagen (usando* `leerPGMbinario()`*).*

Ejercicio 3 *Crear la biblioteca* `libimagen.a` *con los ficheros* `imagen.o` *y* `pgm.o`*.*

Ejercicio 4 *Probar la clase compilando y ejecutando el programa* testimagen.cpp *proporcionado en el directorio* `src`*. Se debe crear un* `makefile` *que compile los fuentes, cree la biblioteca y cree el ejecutable. El resultado de ejecutar bin/testimage será:*

```
$ bin/testimagen
degradado.pgm guardado correctamente
usa: display degradado.pgm para ver el resultado
trozo.pgm guardado correctamente
usa: display trozo.pgm para ver el resultado
```

degradado.pgm

trozo.pgm

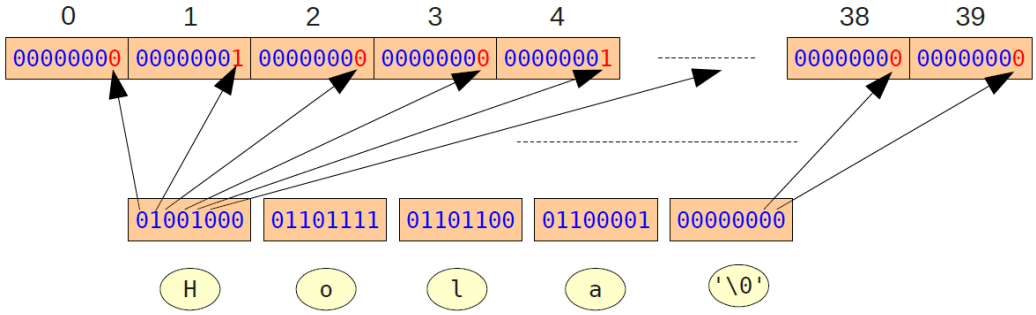
4. Ocultar/Revelar un mensaje

La técnica que vamos a utilizar para ocultar (y posteriormente extraer) un mensaje de texto en la imagen consiste en modificar el bit menos significativo[3] de cada píxel de la imagen para introducir un bit del mensaje.

[3] El bit más a la derecha, y que corresponde a las unidades del número binario.

Nótese que haciendo esto afectaremos el valor del píxel, como mucho, en una unidad de 255. La imagen la veremos, por tanto, prácticamente igual.

Si el mensaje está almacenado en una cadena-C, es decir, una secuencia de valores de tipo char que terminan en un cero, podemos "repartir" cada carácter (8 bits) del mensaje en 8 píxeles consecutivos. En la siguiente figura mostramos un esquema que refleja esta idea:



La secuencia de 40 bytes en la fila superior corresponde a los valores almacenados en el vector de "byte" que corresponde a la imagen. En la fila inferior, podemos ver un mensaje con 4 caracteres (5 incluyendo el carácter cero final) que corresponde al mensaje a ocultar. Observe cómo los bits del mensaje 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 realizar la operación inversa, es decir, consultar cada uno de los bits menos significativos y colocarlos de forma consecutiva, creando una secuencia de 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.

Ejercicio 5 *Cree los archivos* `byte.h` *y* `byte.cpp` *copiando el módulo* `bloqueLed` *definido en la practica 2, y reemplazando los términos "bloqueLed" y "LED" por "byte" y "bit", respectivamente. De esta forma, se podrán utilizar las operaciones definidas para manipular LEDs en* bloqueLed *para manipular bits en los bytes que componen una cadena-C y una imagen.*

Ejercicio 6 *Crear una función "*ocultar*", en el módulo* `codificar` *(compuesto por* `codificar.h` *y* `codificar.cpp`*), que recibe un objeto de la clase* Imagen *y una cadena-C con el mensaje a ocultar, e inserta dicho mensaje en la imagen recibida. La imagen modificada, será visualmente similar a la imagen de entrada, pero ocultará la cadena insertada.*

La función "`ocultar`*" deberá considerar posibles situaciones de error y devolver un valor que indique si se ha conseguido realizar la operación con éxito o ha habido algún error. Un ejemplo de una posible situación de error es que la cadena que se intenta codificar sea demasiado grande para la imagen dada.*

Ejercicio 7 *Crear una función "*`revelar`*", en el módulo* `codificar`*, que recibe como entrada un objeto de la clase* Imagen *en el cual se ha ocultado previamente un mensaje y devuelve una cadena-C con el mensaje oculto en dicha imagen.*

La función "`revelar`*" también deberá considerar posibles situaciones de error y devolver un valor que indique si se ha conseguido realizar la operación con éxito o ha habido algún error. Ejemplos de posibles situaciones de error son:*

- *La imagen no contiene ningún mensaje, es decir, no se encuentra ningún carácter terminador de cadena (carácter '*`\0`*')).*
- *La imagen contiene una cadena de tamaño mayor que el de la cadena-C que se pasa como parámetro a la función "*`revelar`*".*

La función programada podrá recibir parámetros adicionales, necesarios para detectar dichos errores (p. e., el tamaño máximo de la cadena-C).

Ejercicio 8 *Modificar el makefile del ejercicio 4 para crear la biblioteca* `libimagen.a` *añadiendo los ficheros* `codificar.o` *y* `byte.o`*.*

Ejercicio 9 *Probar las funciones compilando y ejecutando el programa* `testcodificar.cpp` *proporcionado en el directorio* `src`*. Se debe modificar el* `makefile` *que compile los fuentes, cree la biblioteca y cree el ejecutable. El resultado de ejecutar* bin/testcodificar *será:*

```
$ bin/testcodificar
La imagen original tiene los siguientes valores
F G H I J
K L M N O
P Q R S T
U V W X Y
Los bits menos significativos de los  primeros 16 pixeles
   de la imagen son
0101010101010101
Voy a ocultar la cadena 'X' y el '\0' en la imagen
La cadena tiene 16 bits que son:
0101100000000000
La imagen después de ocultar tiene los siguientes valores
F G H I K
J L L N N
P P R R T
T V W X Y
Los bits menos significativos de los 16 primeros pixeles de
    la imagen son
0101100000000000
 y deberían ser:
0101100000000000
El mensaje oculto en la imagen es: 'X'
Voy a ocultar la cadena 'Supercalifragilisticoespialidoso'
   y el '\0' en la imagen
Error: El mensaje es muy largo para ocultarlo en la imagen
```

Ejercicio 10 *Implementar un único programa "*`esteganografia`*" que contenga las llamadas a las funciones "*ocultar*" y "*revelar*". Para ello, se deberá generar un menú inicial con las opciones 1 - "Ocultar", 2- "Revelar" y 3- "Salir". De acuerdo a la opción elegida, el programa solicitará los parámetros correspondientes a dicha opción y ejecutara la función seleccionada. Un ejemplo de ejecución con la opción "Ocultar" podría ser el siguiente:*

```
prompt% esteganografia <enter>
1 - Ocultar
2 - Revelar
3 - Salir

Seleccione una opcion: 1 <enter>

Introduzca la imagen de entrada: lena.pgm
Introduzca la imagen de salida: salida.pgm
Introduzca el mensaje: ¡Hola mundo!

Ocultando...
prompt%
```

Un ejemplo de ejecución con la opción "Revelar" podría ser el siguiente:

```
prompt% esteganografia <enter>
1 - Ocultar
2 - Revelar
3 - Salir

Seleccione una opcion: 2 <enter>

Introduzca la imagen de entrada: salida.pgm

Revelando...

El mensaje obtenido es: ¡Hola mundo!

prompt%
```

Lógicamente, esta ejecución corresponde a un caso con éxito, ya que si ocurre algún tipo de error, deberá acabar con un mensaje de error adecuado.

En el paquete suministrado se proporciona una serie de imágenes de prueba. Algunas de ellas pueden llevar ya un mensaje oculto.

Los nombres de las imágenes de entrada y de salida ("lena.pgm" , "salida.pgm" en los ejemplos) deberán estar definidos en cadenas-C e implementados de la siguiente forma:

```
const int MAXNOMBRE = 100;
char nombre_imagen[MAXNOMBRE];
```

El mensaje a insertar ("¡Hola mundo!" en el ejemplo) estará definido en una cadena-C, y su correcta lectura requerirá que dicha cadena sea leída hasta el final de línea (utilizar la instrucción `cin.getline()`*). El mensaje puede definirse como*

```
const int MAXTAM = 256;
char mensaje[MAXTAM];
```

Ejercicio 11 *Añadir al makefile del ejercicio 8 todo lo necesario para poder compilar el programa "esteganografia".*

5. Material a entregar

Cuando esté todo listo y probado el alumno empaquetará la estructura de directorios anterior en un archivo con el nombre **esteganografia.zip** y lo entregará en la plataforma **decsai** en el plazo indicado. No deben entregarse archivos objeto (**.o**) ni ejecutables. Para asegurarse de esto último conviene ejecutar **make clean** antes de proceder al empaquetado.

El alumno debe asegurarse de que ejecutando las siguientes órdenes se compila y ejecuta correctamente su proyecto:

```
unzip estenografia.zip
cd estenografia
make
bin/estenografia
```