Mensajes e 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 el uso de "*arrays*".
2. Usar tipos de datos "*enum*" (enumerados).
3. Practicar con operaciones a nivel de bit.

Los requisitos para poder realizar esta práctica son:

1. Saber manejar los *arrays* así como el caso particular de *cadenas-C* .
2. Conocer el diseño de programas en módulos independientes, así como la compilación separada, incluyendo la creación de bibliotecas.
3. Conocer el diseño de ficheros *makefile*.

El alumno debe realizar esta práctica una vez que haya estudiado los contenidos sobre *arrays* y *cadenas-C*. No será necesario -de hecho estará prohibido- usar tipos puntero o memoria dinámica, e incluso los tipos de la STL, como *vector<>* o *string*. En esta sección se completan los conocimientos necesarios para poder realizar la práctica.

1.1. Tipo "*Enumerado*"

En muchos casos, es necesario manejar un tipo de objeto que puede tener un conjunto finito y pequeño de posibles valores. Algunos ejemplos son:

- Día de la semana: con 7 posibles valores, de lunes a domingo.
- Mes del año: con 12 valores desde enero a diciembre.
- Palos de la baraja española: con valores oros, copas, espadas y bastos.

Para estos casos, una solución directa y muy sencilla es el uso de un tipo de dato entero predefinido. Podemos usar, por ejemplo, los valores de 0 a 6 para indicar un día de la semana, de 1 a 12 para un mes, o de 1 al 4 para los palos de la baraja.

Sin embargo, resulta mucho más cómodo y legible usar un tipo de dato que refleje mejor el tipo de objeto que se maneja, así como sus posibles valores. Para ello, el lenguaje nos permite **crear** nuevos tipos de datos enumerados -pues se crean indicando todos y cada uno de sus valores- con la palabra reservada "*enum*". La sintaxis para declarar el nuevo tipo es:

```
enum <nombre del tipo> { <lista de posibles valores> }
```

Por ejemplo, se podrían crear dos tipos de datos *DiaSemana* y *Mes*, de forma que sea más legible el código que maneja este tipo de objetos. Por ejemplo, podemos obtener el siguiente código:

```
enum DiaSemana {LUNES,MARTES,MIERCOLES,JUEVES,VIERNES,SABADO,DOMINGO};
enum Mes {ENERO,FEBRERO,MARZO,ABRIL,MAYO,JUNIO,
          JULIO,AGOSTO,SEPTIEMBRE,OCTUBRE,NOVIEMBRE,DICIEMBRE};

DiaSemana ObtenerDiaSemana(int d, Mes m, int a) {
    // ... código....
}

DiaSemana cae_en= ObtenerDiaSemana(25,DICIEMBRE,2011);
if (cae_en==DOMINGO)
  cout << "La navidad cae en domingo..." << endl;
```

La forma en que el compilador maneja estos nuevos tipos internamente es muy simple, ya que prácticamente lo que hace es considerar que los nuevos tipos almacenan algún tipo de entero y que cada uno de los valores concretos que se enumeran representan una constante entera (de ahí que los hayamos definido con todas las letras en mayúscula).

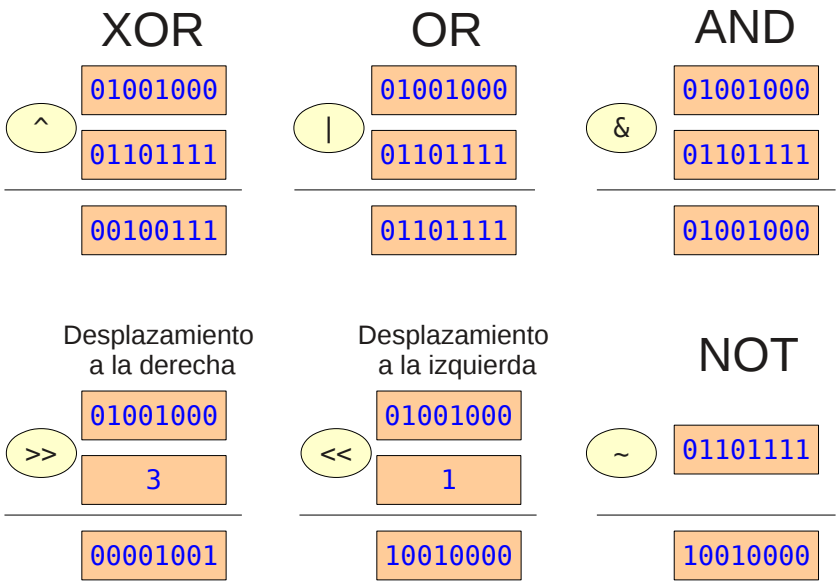
Para más información véase [CPROGa] o página 179 y siguientes de [GAR06a].

1.2. Operadores a nivel de bit

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.
  - No (~exp1). Obtiene un nuevo valor con los bits cambiados, es decir, ceros por unos y unos por ceros.

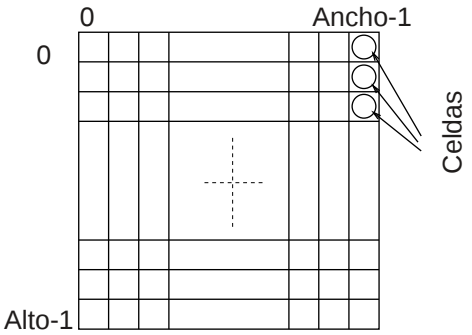
En la siguiente figura se muestran algunos ejemplos con los operadores que hemos indicado.



Con estos operadores, se pueden crear expresiones para consultar el valor de los bits que componen un dato, o para modificarlos.

2. Imágenes

Desde un punto de vista práctico, una imagen se puede considerar como un conjunto de celdas que se organizan en posiciones que podemos hacer corresponder con una matriz bidimensional tal como muestra la siguiente figura.



El contenido de cada una de las celdas (que llamaremos también píxel) dependerá en gran medida de la aplicación donde se quiera utilizar. Algunos ejemplos podrían ser:

- Una imagen para señalar los puntos donde se puede encontrar información de un objeto en otra imagen, es decir, uma imagen para la que queremos guardar una información binaria para cada punto. En este caso, bastaría con almacenar un bit en cada una de las celdas.
- Si queremos almacenar una escena en blanco y negro, podemos crear un rango de valores de luminosidad (que llamaremos a partir de ahora valores de gris), por ejemplo los enteros en el rango [0,255] (el cero es negro, y el 255 blanco). En este caso, cada celda puede almacenar un único byte.
- Si queremos almacenar una imagen médica, donde cada punto mantiene la densidad obtenida a partir de un aparato de rayos X, el rango de posibles valores podría estar en [0,4096] y, por tanto, cada celda almacenaría un valor de este rango (por ejemplo, un entero).
- Si queremos almacenar una escena con información de color, podemos fijar en cada celda una tripleta de valores indicando el nivel de intensidad con el que contribuyen 3 colores básicos para formar el color requerido.

En la práctica que se propone en este documento, trabajaremos con imágenes en blanco y negro y en color.

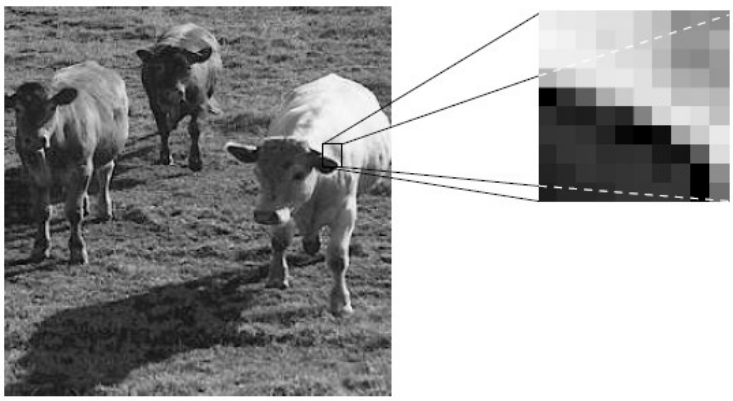
2.1. Imágenes en blanco y negro

Para representar las imágenes en blanco y negro podemos usar un rango de valores para indicar todas las tonalidades de gris que van desde el negro hasta el blanco. En nuestro caso, las imágenes almacenarán en cada *píxel* un valor de gris desde el 0 al 255. Por ejemplo, un píxel con valor 128 tendrá un gris intermedio entre blanco y negro.

La elección del rango [0,255] se debe a que esos valores son los que se pueden representar en un *byte*[1](8 *bits*). Por tanto, si queremos almacenar una imagen de niveles de gris, necesitaremos *ancho·alto* bytes. En el ejemplo de la imagen anterior, necesitaríamos 64Kbytes para representar todos sus píxeles.

En la siguiente figura se muestra un ejemplo de imagen 256x256 de niveles de gris. Observe el zoom de una región 10x10 para apreciar con detalle los grises que la componen.

1 *Recuerde que en nuestro caso, el tipo más adecuado para almacenar este rango es "unsigned char".*



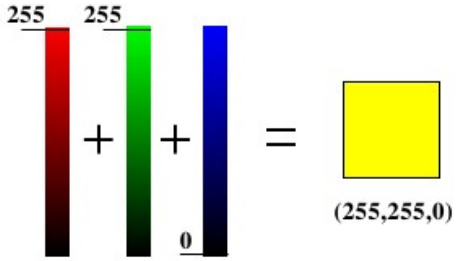
2.2. Imágenes en color

Para representar un color de forma numérica, no es fácil usar 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, generalmente adaptadas a diferentes objetivos y necesidades.

En una imagen en color, el contenido de cada *píxel* será una tripleta de valores según un determinado modelo de color. En esta práctica consideraremos 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[2].

Podemos considerar distintas alternativas para el rango de posibles valores de cada componente, aunque en la práctica, es habitual asignarle el rango de números enteros desde el 0 al 255, ya que permite representar cada componente con un único *byte*, y la variedad de posibles colores es suficientemente amplia. Por ejemplo, el ojo humano no es capaz de distinguir un cambio de una unidad en cualquiera de las componentes.

En la siguiente figura 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.



2.3. Funciones de E/S de imágenes

Las imágenes que manejaremos están almacenadas en un fichero que se divide en dos partes:

1. *Cabecera*. En esta parte se incluye información acerca de la imagen, sin incluir el valor de ningún píxel concreto. Así, podemos encontrar valores que indican el tipo de imagen que es, comentarios sobre la imagen, el rango de posibles valores de cada píxel, etc. En esta práctica, esta parte nos va a permitir consultar el tipo de imagen y sus dimensiones sin necesidad de leerla.
2. *Información*. Contiene los valores que corresponden a cada píxel. Hay muchas formas para guardarlos, dependiendo del tipo de imagen de que se trate, pero en nuestro caso será muy simple, ya que se guardan todos los bytes por filas, desde la esquina superior izquierda a la esquina inferior derecha.

2 *Red, Green, Blue.*

Los tipos de imagen que vamos a manejar serán *PGM* (*Portable Grey Map file format*) y *PPM* (*Portable Pix Map file format*), que tienen un esquema de almacenamiento con cabecera seguida de la información, como hemos indicado. El primero se usará para las imágenes en blanco y negro y el segundo para las imágenes en color.

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 de ambos formatos. 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. El archivo de cabecera contiene lo siguiente:

```
#ifndef _IMAGEN_ES_H_
#define _IMAGEN_ES_H_

enum TipoImagen {IMG_DESCONOCIDO, ///< Tipo de imagen desconocido
                 IMG_PGM,         ///< Imagen tipo PGM
                 IMG_PPM          ///< Imagen tipo PPM
                };
TipoImagen LeerTipoImagen (const char nombre[], int& filas,
                           int& columnas);
bool LeerImagenPPM (const char nombre[], int& filas, int& columnas,
                    unsigned char buffer[]);
bool EscribirImagenPPM (const char nombre[], const unsigned char datos[],
                        int f, int c);
bool LeerImagenPGM (const char nombre[], int& filas, int& columnas,
                    unsigned char buffer[]);
bool EscribirImagenPGM (const char nombre[], const unsigned char datos[],
                        int f, int c);

#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).

Si estudia detenidamente las cabeceras de las funciones que se proporcionan, verá que con el nombre del parámetro, así como con su carácter de entrada o salida, es fácil intuir el objetivo de cada uno de ellas. Tal vez, la parte más confusa pueda surgir en los parámetros correspondientes al buffer o los datos de la imagen (vectores de *unsigned char*):

1. Si la imagen es *PGM* -de grises- será un vector que contenga todos los *bytes* consecutivos de la imagen. Así, la posición 0 del vector tendrá el píxel de la esquina superior izquierda, la posición 1 el de su derecha, etc.
2. Si la imagen es *PPM* -de color- será un vector similar. En este caso, la posición 0 tendrá la componente R de la esquina superior izquierda, la posición 1 tendrá la poción G, la posición 2 la B, la posición 3 la componente R del siguiente píxel, etc. Es decir, añadiendo las tripletas *RGB* de cada píxel.

3. Ocultar/Revelar un mensaje

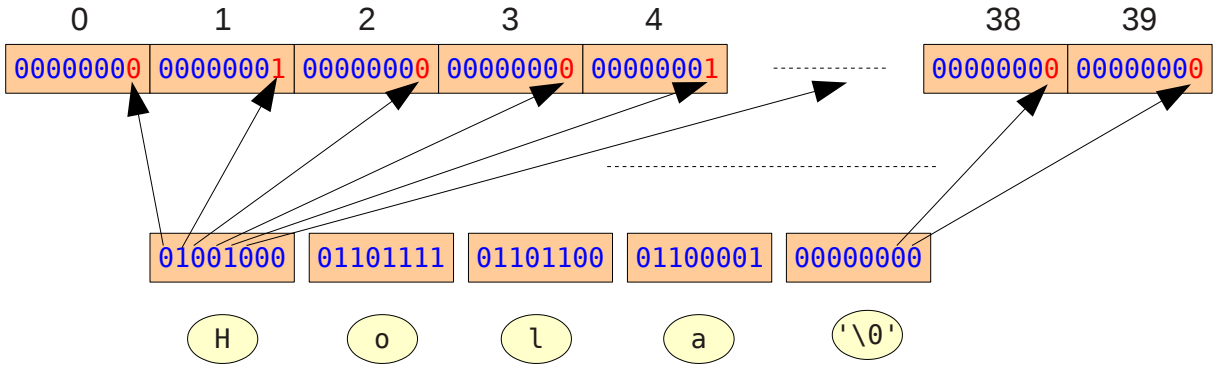
La práctica va a consistir, en la inserción y extracción de un mensaje "oculto" en una imagen. Para ello, modificaremos el valor de cada píxel para que contenga parte de la información a ocultar. Ahora bien, ¿Cómo almacenamos un mensaje (*cadena-C*) dentro de una imagen?

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[3], 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) superior corresponde a los valores almacenados en el vector de "*unsigned char*" que corresponde a la imagen. Podemos suponer, por ejemplo, que la imagen es negra, y que por tanto todos los píxeles tienen un valor de cero.

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

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

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

4. Desarrollo de la práctica

Para resolver este problema, el alumno tendrá que llevar a cabo una serie de tareas, que exponemos en esta sección, junto con las condiciones o restricciones que deberá tener en cuenta.

Antes de comenzar, el alumno debe descargar un paquete con el material y datos básicos a partir del cual desarrollar la práctica. En este paquete, por ejemplo, encontrará el código que resuelve el problema de la E/S de imágenes, así como alguna imagen de ejemplo.

4.1. Módulo codificar

La tarea básica que hay que realizar en la práctica consiste en la inserción y extracción de u 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.

---

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

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.

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 lo considera oportuno, puede 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 puede optar por imponer precondiciones a esas funciones, en cuyo caso, debería comprobar que no hay errores -se cumplen las condiciones- en el lugar de la llamada

El alumno debe crear los archivos “*codificar.h*” y “*codificar.cpp*” para resolver estos dos problemas. Tenga en cuenta que puede incluir la devolución de algún valor que indique si se ha conseguido realizar la operación con éxito.

4.2. Documentación

Si revisa el material que se ha descargado para desarrollar la práctica, encontrará que el módulo de E/S de imágenes está documentado en base a la sintaxis de doxygen. De hecho, incluso se ha proporcionado lo necesario para poder realizar fácilmente la generación de la documentación asociada en formato *html*.

De igual forma, el alumno debe añadir la documentación del módulo que ha desarrollado, más concretamente, añadir comentarios *doxygen* al fichero “*codificar.h*” que ha creado.

4.3. Programas

El objetivo final de la práctica es crear dos programas, uno para ocultar un mensaje en una imagen y otro para revelarlo.

4.3.1. Ocultar

El programa 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%
```

donde hemos resaltado en color rojo los datos que introduce el usuario desde el teclado. 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!*”.

Observe 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.

4.3.2. 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
Introduzca la imagen de entrada: salida.ppm
Revelando...
El mensaje obtenido es:
¡Hola mundo!
prompt%
```

donde hemos resaltado en color rojo los datos introducidos por el usuario. Observe 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, tenga en cuenta que si la ejecución encuentra un error, deberá terminar con el mensaje correspondiente.

4.3.3. Restricciones

Para resolver estos problemas, será necesario cargar en memoria imágenes (*arrays* de *bytes)* y cadenas de caracteres. En la solución que proponga, no se podrá usar memoria dinámica ni punteros, es decir, será necesario que declare los vectores que necesite como *arrays* de tamaño fijo, ya sea para almacenar los *bytes* de la imagen o los *char* del mensaje.

Como ejemplo, si deseamos cargar una imagen con un tamaño máximo de 1.000.000, podemos incluir el siguiente código:

```
int main()
{
  const int MAXBUFFER= 1000000;
  const int MAXNOMBRE= 100;
  char nombre_imagen[MAXNOMBRE];
  unsigned char buffer[MAXBUFFER];
```

donde puede ver que también hemos indicado una *cadena-C* (para el nombre de la imagen) que contendrá como mucho 100 caracteres. Para esta práctica, podemos limitar el tamaño máximo de la imagen a 1.000.000 *bytes*, el nombre de una imagen a 100 *bytes*, y el tamaño máximo de un mensaje a 125.000 *bytes*.

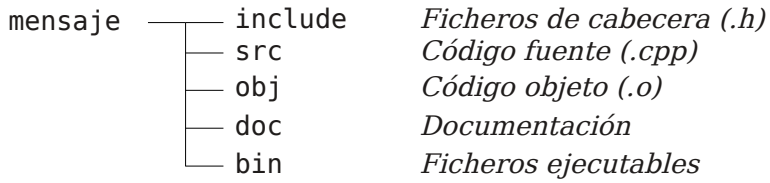
Lógicamente, cuando indicamos 1.000.000, nos referimos al número total de bytes que ocupa la imagen, ya sea en grises o en color. Por ejemplo, una imagen 1.000x1.000 de grises ocuparía el 100% de ese espacio, mientras que una 1.000x1.000 en color no cabe, ya que necesitaría el triple de espacio, al requerir 3 *bytes* para cada píxel.

4.4. Práctica a entregar

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

```
prompt% tar zcvf mensaje.tgz mensaje
```

tras lo cual, dispondrá de un nuevo archivo *mensaje.tgz* que contiene la carpeta mensaje, 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.
- [STR02] Stroustrup, B. "*El lenguaje de programación C++*". Edición Especial. Addison Wesley, 2002.
- [CPROGa] "*Enumerated Types – enums*". Tutorial disponible en internet en la dirección: "http://www.cprogramming.com/tutorial/enum.html".
- [CPROGb] "Bitwise Operators in C and C++: A Tutorial". Disponible en internet en la dirección: "http://www.cprogramming.com/tutorial/bitwise_operators.html".