

MANEJO DE FICHEROS BMP

typedef unsigned short int WORD; // 16 bits o 2 bytes

typedef unsigned char BYTE; // 8 bits

typedef unsigned long DWORD; // 32 bits o 4 bytes

Números enteros sin signo

- Tipos "unsigned" de C
- Representan solamente la magnitud, en forma binaria.
Permiten almacenar números desde 0 hasta $(2^n) - 1$
n se refiere al número de bits usados para representar
- Números de 1 byte sin signo: 0 a 255: *unsigned char*
- Números de 2 bytes sin signo: 0 a 65535: *unsigned short*
- Números de 4 bytes sin signo: 0 a algo más de 4 mil millones: *unsigned long*

La siguiente tabla muestra todas las combinaciones que se ajustan al estándar ANSI junto con sus rangos mínimos y longitudes aproximadas en bits.

Tipo:	Tamaño en bits	Rango
char:	8	-127 a 127
unsigned char:	8	0 a 255
signed char:	8	-127 a 127
int:	16	-32767 a 32767
unsigned int:	16	0 a 65535
signed int:	16	-32767 a 32767
short int:	16	-32767 a 32767
unsigned short int:	16	0 a 65535
signed short int:	16	-32767 a 32767
long int:	32	-2147483647 a 2147483647
signed long int:	32	-2147483647 a 2147483647
unsigned long int:	32	0 a 4294967295
float:	32	seis dígitos de precisión
double:	64	diez dígitos de precisión
long double:	64	diez dígitos de precisión

1 Bit = 0,1

1 Byte = 8 Bits

1Mb = 1024 Bytes

16 Bits = 1 Word

32 Bits = Double Word (Procesadores 80386)

El formato BMP (Windows BitMaP) es probablemente el formato de fichero para imágenes más simple que existe. Aunque teóricamente permite compresión, en la práctica nunca se usa, y consiste simplemente en una cabecera y a continuación los valores de cada pixel, comenzando por la línea de más abajo y terminando por la superior, pixel a pixel de izquierda a derecha. Parece ser el formato preferido por Bill Gates.

Su única ventaja es su sencillez. Su gran desventaja es el enorme tamaño de los ficheros.

Un ejemplo en Turbo Pascal.

```
TYPE BitMapHeader=RECORD
  {BitMap File Header}
    bfType: INTEGER;
    bfSize: LONGINT;
    bfReserved1: INTEGER;
    bfReserved2: INTEGER;
    bfOffbits: LONGINT;
  {BitMap Info Header}
    biSize: LONGINT;
    biWidth: LONGINT;
    biHeight: LONGINT;
    biPlanes: INTEGER;
    biBitCount: INTEGER;
    biCompression: LONGINT;
    biSizeImage: LONGINT;
    biXpelsPerMeter: LONGINT;
    biYpelsPerMeter: LONGINT;
    biClrUsed: LONGINT;
    biClrImportant: LONGINT;
END;
```

Tipo	Rango	Formato
byte	0 .. 255	8 bits sin signo
integer	-32768 .. 32767	16 bits con signo
longint	-247483648 .. 2147483647	32 bits con signo
shortint	-128 .. 127	8 bits con signo
word	0 .. 65535	16 bits sin signo

Manejo de ficheros BMP

Los archivos BMP, tienen las siguientes partes:

1. **Encabezado:** es la parte que contiene información sobre archivo.
2. **Cuerpo:** es la parte que contiene los datos a utilizar del archivo. En el caso de los ficheros BMP, el cuerpo se corresponde con la descripción RGB de los píxeles de la imagen.

Tipo BITMAPINFOHEADER

The **BITMAPINFOHEADER** structure contains information about the dimensions and color format of device-independent bitmaps (DIB).

```
typedef struct tagBITMAPINFOHEADER{
    DWORD biSize;
    LONG biWidth;
    LONG biHeight;
    WORD biPlanes;
    WORD biBitCount;
    DWORD biCompression;
    DWORD biSizeImage;
    LONG biXPelsPerMeter;
    LONG biYPelsPerMeter;
    DWORD biClrUsed;
    DWORD biClrImportant;
} BITMAPINFOHEADER, *PBITMAPINFOHEADER;
```

Campos de Tipo BITMAPINFOHEADER

biSize Specifies the number of bytes required by the structure.

biWidth Specifies the width of the bitmap, in pixels.

biHeight Specifies the height of the bitmap, in pixels. If **biHeight** is positive, the bitmap is a bottom-up DIB and its origin is the lower-left corner. If **biHeight** is negative, the bitmap is a top-down DIB and its origin is the upper-left corner.

biPlanes Specifies the number of planes for the target device. This value must be set to 1.

biBitCount Specifies the number of bits-per-pixel. The **biBitCount** member of the **BITMAPINFOHEADER** structure determines the number of bits that define each pixel and the maximum number of colors in the bitmap.

biCompression Specifies the type of compression for a compressed bottom-up bitmap (top-down DIBs cannot be compressed).

biSizeImage Specifies the size, in bytes, of the image. This may be set to zero for BI_RGB bitmaps.

biXPelsPerMeter Specifies the horizontal resolution, in pixels-per-meter, of the target device for the bitmap. An application can use this value to select a bitmap from a resource group that best matches the characteristics of the current device.

biYPelsPerMeter Specifies the vertical resolution, in pixels-per-meter, of the target device for the bitmap.

biClrUsed Specifies the number of color indexes in the color table that are actually used by the bitmap. If this value is zero, the bitmap uses the maximum number of colors corresponding to the value of the **biBitCount** member for the compression mode specified by **biCompression**.

biClrImportant Specifies the number of color indexes that are required for displaying the bitmap. If this value is zero, all colors are required.

CUERPO DE LOS FICHEROS BITMAP

El cuerpo de un fichero bitmap contiene la descripción de los colores de todos los píxeles de la imagen. El valor de cada color RGB de un píxel se corresponde con un dato del tipo **unsigned char** (lo que se corresponde con 1 byte, que a su vez significa 256 valores diferentes). El primer byte del cuerpo se corresponde con la cantidad de **azul** que tiene el primer píxel de la imagen (para saber cuál es el primer píxel de una imagen lea la descripción del campo **biheight** de la estructura **tagBITMAPINFOHEADER**) El segundo byte del cuerpo se corresponde con la cantidad de **verde** del mismo píxel y el tercer byte se corresponde con la cantidad de **rojo**. Con lo cual con los tres primeros bytes se describe el color RGB del primer píxel. A continuación los siguientes 3 bytes (el 4º, 5º y 6º) describen el RGB del 2º píxel de la imagen y así sucesivamente hasta recorrer los **biWidth X biHeight** píxeles que componen la imagen. Para leer y escribir sobre los colores de los píxeles de la imagen nos valdremos de las funciones **fgetc** y **fputc** respectivamente y junto con un puntero al fichero.

Lectura de una imagen BMP y escritura del negativo de ésta en un segundo fichero BMP trabajando con vectores.

A continuación se presenta un ejemplo de cómo leer correctamente un fichero BMP y de cómo crear otro fichero BMP con el negativo de la imagen anterior. En este ejemplo, la lectura se hace leyendo los datos y guardándolos en 3 vectores (AZUL, VERDE y ROJO).

```
#include <windows.h>
#include <iostream.h>
#include <stdio.h>
void main(void)
{
    long altura, anchura;
    char ArchivoOrigen[100], ArchivoDestino[100];
    // Estructuras de datos para almacenar los encabezamientos del BMP
    BITMAPFILEHEADER EncabezadoImagen;
    BITMAPINFOHEADER InformacionEncabezadoImagen;
    FILE *PunteroImagenInicial;
    FILE *PunteroImagenFinal;
    while (true)
    {
        cout<<"Escriba el nombre del archivo de origen: ";
        cin.getline(ArchivoOrigen,100);
        // Abre el archivo indicado
        PunteroImagenInicial = fopen(ArchivoOrigen,"rb");
        if (PunteroImagenInicial==0)
            cout<<"Ese archivo no existe."<<endl;
        else
            break;
    }
    cout<<"Escriba el nombre del archivo de destino: ";
    cin.getline(ArchivoDestino,100);
    // Abre el archivo prepara el archivo indicado para que sea escrito
    PunteroImagenFinal = fopen(ArchivoDestino, "wb");
    // Leemos los encabezamientos
    fread(&EncabezadoImagen,sizeof(EncabezadoImagen), 1, PunteroImagenInicial);
    fread(&InformacionEncabezadoImagen,sizeof(InformacionEncabezadoImagen), 1,
    PunteroImagenInicial);
    //Escribimos los encabezados en el fichero final
    fwrite(&EncabezadoImagen,sizeof(EncabezadoImagen),1,PunteroImagenFinal);
    fwrite(&InformacionEncabezadoImagen,sizeof(InformacionEncabezadoImagen),1,PunteroImagenFinal);
    //Guardamos la anchura y altura de la imagen
    altura =InformacionEncabezadoImagen.biHeight;
    anchura=InformacionEncabezadoImagen.biWidth;
    unsigned char *Rojo=new unsigned char[altura*anchura];
    unsigned char *Verde=new unsigned char[altura*anchura];
    unsigned char *Azul=new unsigned char[altura*anchura];
    // Leemos la Imagen Inicial
    for(int i=0 ; i<(altura * anchura) ; ++i)
    {
        Azul[i] = fgetc(PunteroImagenInicial); // B
        Verde[i] =fgetc(PunteroImagenInicial); // G
```

```
        Rojo[i] =fgetc(PunteroImagenInicial); // R
    }
    //Escribimos sobre la Imagen Final
    for(i=0 ; i<(altura * anchura) ; ++i)
    {
        fputc(255-Azul[i],PunteroImagenFinal);
        fputc(255-Verde[i],PunteroImagenFinal);
        fputc(255-Rojo[i],PunteroImagenFinal);
    }
    // Cerramos el archivo
    fclose(PunteroImagenInicial);
    fclose(PunteroImagenFinal);
    cout<<"Se ha generado el archivo;"<<endl;
}
}
```

Lectura de una imagen BMP y escritura del negativo de ésta en un segundo fichero BMP trabajando con matrices.

Para trabajar con imágenes, hay veces que no interesa tener los datos guardados de manera vectorial, sino de forma matricial. En el caso de la obtención del negativo de la imagen, evidentemente, lo mejor es tener todos los datos guardados en un vector. Pero, a veces, se quieren realizar operaciones bidimensionales sobre la imagen, como puede ser un filtrado de ruido, una transformación geométrica, etc. En estos casos, sí que interesa tener en cuenta la fila y columna en la que nos encontramos. Por esta razón, vamos a presentar el mismo programa anterior, pero guardando los datos en tres matrices (ROJO, VERDE y AZUL).

```
#include <windows.h>
#include <iostream.h>
#include <stdio.h>
void main(void)
{
    long altura, anchura;
    unsigned int f,c;
    char ArchivoOrigen[100],ArchivoDestino[100];
    BITMAPFILEHEADER EncabezadoImagen;
    BITMAPINFOHEADER InformacionEncabezadoImagen;
    // Puntero a estructuras de datos del tipo FILE
    FILE *PunteroImagenInicial;
    FILE *PunteroImagenFinal;
    while (true)
    {
        cout<<"Escriba el nombre del archivo de origen: ";
        cin.getline(ArchivoOrigen,100);
        // Abre el archivo indicado
        PunteroImagenInicial = fopen(ArchivoOrigen,"rb");
        if (PunteroImagenInicial==0)
            cout<<"Ese archivo no existe."<<endl;
        else
            break;
    }
    cout<<"Escriba el nombre del archivo de destino: ";
```

```
cin.getline(ArchivoDestino,100);
// Abre el archivo prepara el archivo indicado para que sea escrito
PunteroImagenFinal = fopen(ArchivoDestino, "wb");
// Leemos los encabezamientos
fread(&EncabezadoImagen,sizeof(EncabezadoImagen), 1, PunteroImagenInicial);
fread(&InformacionEncabezadoImagen,sizeof(InformacionEncabezadoImagen), 1,
PunteroImagenInicial);
//Escribimos los encabezados en el fichero final
fwrite(&EncabezadoImagen,sizeof(EncabezadoImagen),1,PunteroImagenFinal);
fwrite(&InformacionEncabezadoImagen,sizeof(InformacionEncabezadoImagen),1,Punt
eroImagenFinal);
//Guardamos la anchura y altura de la imagen
altura =InformacionEncabezadoImagen.biHeight;
anchura=InformacionEncabezadoImagen.biWidth;
unsigned char **Rojo=new unsigned char*[altura];
for(f=0;f<altura;f++)
{
    *(Rojo+f)=new unsigned char[anchura];
}
unsigned char **Verde=new unsigned char*[altura];
for(f=0;f<altura;f++)
{
    *(Verde+f)=new unsigned char[anchura];
}
unsigned char **Azul=new unsigned char*[altura];
for(f=0;f<altura;f++)
{
    *(Azul+f)=new unsigned char[anchura];
}
// Leemos la Imagen Inicial
for(f=0 ; f<altura; ++f)
{
    for (c=0;c<anchura;c++)
    {
        Azul[f][c] = fgetc(PunteroImagenInicial); // B
        Verde[f][c] =fgetc(PunteroImagenInicial); // G
        Rojo[f][c] =fgetc(PunteroImagenInicial); // R
    }
}
//Escribimos sobre la Imagen Final
for(f=0 ; f<altura; ++f)
{
    for (c=0;c<anchura;c++)
    {
        fputc(255-Azul[f][c],PunteroImagenFinal);
        fputc(255-Verde[f][c],PunteroImagenFinal);
        fputc(255-Rojo[f][c],PunteroImagenFinal);
    }
}
fclose(PunteroImagenInicial);
fclose(PunteroImagenFinal);
cout<<"Se ha generado el archivo;"<<endl;
}
```

Lectura de una imagen BMP y escritura de ésta en escala de grises en un segundo fichero BMP trabajando con vectores.

Lectura de una imagen BMP y escritura de ésta modificando el brillo en un segundo fichero BMP trabajando con matrices.

Lectura de una imagen BMP y escritura de ésta rotada 90° en el sentido de la agujas del reloj en un segundo fichero BMP trabajando con subfunciones.

En este ejercicio se propone realizar un programa que abra una imagen BMP y cree un nuevo archivo BMP con la imagen girada 90° en el sentido de las agujas del reloj. Para realizar esta transformación geométrica lo más interesante es almacenar los colores RGB aportados por el archivo de origen en tres matrices (**Rojal**, **Verdel**, **Azul**). Y posteriormente guardar los datos de estas tres matrices en otras tres matrices finales (**RojaF**, **VerdeF**, **AzulF**) donde se cumpla que para todo par de valores $[i, j]$ entonces:

Rojal $[i][j] = \text{RojaF } [j][i]$

Verdel $[i][j] = \text{VerdeF } [j][i]$

Azul $[i][j] = \text{AzulF } [j][i]$

Aunque el ejercicio es realmente sencillo, vamos a tratar de simplificarlo más todavía y nos centraremos en resolver este caso cuando la imagen que tratamos sea cuadrada. El que la imagen sea cuadrada trae consigo que no hay que modificar el encabezado del archivo BMP destino, sino que basta que este encabezado sea el mismo que el del archivo origen, cosa que no sucedería si la imagen fuera rectangular, evidentemente.

No obstante observará que, durante la programación, se pide tener en cuenta el número de píxeles horizontales y verticales, aunque estos vayan a ser iguales en este caso. Esto se hace para que el programa sea fácil de preparar el en caso de querer trabajar con imágenes rectangulares. Bastará entonces con modificar la parte de nuestro programa referente a la preparación del encabezado del "fichero de destino". Además vamos a aprovechar para realizar unas subfunciones que utilizaremos desde la función **main()**. Estas subfunciones serán las siguientes:

unsigned char NuevaMatriz(long F, long C)**

Esta función deberá realizar una reserva dinámica de memoria para almacenar una matriz de **F** filas y **C** columnas. Devolverá, lógicamente, un puntero a **unsigned char** apuntando al primer puntero del vector de punteros que apuntan a las distintas filas de la matriz.

La utilizaremos desde la función main para: crear las 6 matrices de colores RGB que se necesitan (**Rojal**, **Verdel**, **Azul**, **RojaF**, **VerdeF**, **AzulF**).

void LecturaCuerpoBitmap(FILE * PF, unsigned char **R, unsigned char **V, unsigned char **A, long F, long C)

Esta función recibirá un puntero (**PF**) a un fichero RGB abierto, las tres matrices (**R**, **V** y **A**) donde se guardarán los colores del fichero RGB, el número de filas (**F**) y de columnas (**C**) de las matrices.

La utilizaremos desde la función *main* para: inicializar las 3 matrices de colores RGB en las que se guardarán los colores de la imagen origen, inicializándose con ella las matrices **Rojal**, **Verdel** y **Azull**.

void Transponer(unsigned char **M, unsigned char **MT, long F, long C);
Esta función inicializará la matriz **MT** como la transpuesta de la matriz **M**. Además recibe el número de filas (**F**) y de columnas (**C**) de las matrices.

La utilizaremos desde la función *main* para: transponer cada una de las 3 matrices de colores RGB en las que se guardan los colores de la imagen origen, inicializando cada una de las tres veces que se utilice- las matrices **RojaF**, **VerdeF** y **AzulF** respectivamente.

void EscrituraCuerpoBitmap(FILE * PF , unsigned char **R, unsigned char **V, unsigned char **A, long F, long C);

Esta función recibirá un puntero a un fichero RGB (**PF**), las tres matrices (**R**, **V** y **A**) donde se guardarán los colores del fichero RGB, el número de filas (**F**) y de columnas (**C**) de las matrices.

La utilizaremos desde la función *main* para: escribir sobre el cuerpo del fichero destino indicando la composición RGB de cada uno de los píxeles de la imagen.