SANTIAGO AYECHU GÁRRIZ

Práctica 5ª: Imagen Bit Map Portable (BMP)

ÍNDICE

Introducción

Dado un programa codificado en C, el objetivo de esta práctica es desarrollar una subrutina en lenguaje ensamblador equivalente a una función de C. El programa trata sobre la creación de imágenes con formato BMP (Bit Map Portable).

Desarrollo

Se muestran cada uno de los ejercicios correspondientes al guion de la práctica.

Programación en C

A continuación, se muestran los módulos fuente llevados a cabo en esta práctica con sus respectivos comentarios en rojo. Más adelante en la sección de 'Historial Comandos GDB + Salida' se verá la depuración de estos programas.

`bitmap_gen_test.c`

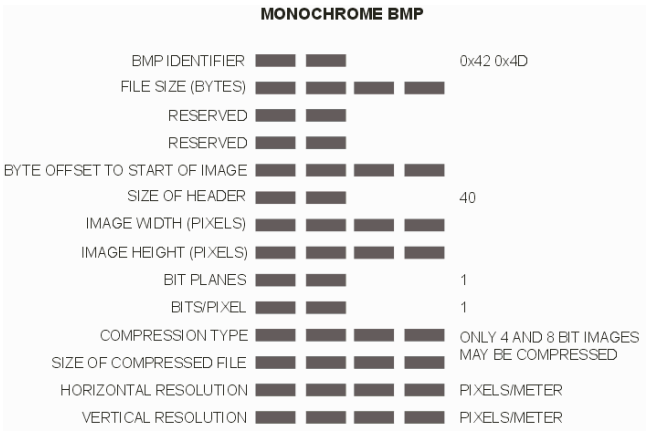
Con este programa, lo que se pretende es realizar un programa el cual crea una imagen en formato BMP de `512x512` y la guarda en un fichero concreto, `test.bmp`.

El formato BMP (Bit Map Portable) es un formato de imagen escalar, esto es, contiene los datos de cada píxel codificando la intensidad de los componentes RGB de color tal como se visualizará en la pantalla.

La pantalla está formada por una matriz bidimensional de pixeles, donde cada píxel es un punto discreto de la pantalla programable. La matriz de la pantalla está vinculada a una estructura de datos tipo array bidimensional 2D de filas (eje horizontal) y columnas (eje vertical) almacenada en la memoria de la tarjeta de video.

El origen de coordenada del array es la esquina inferior izquierda. A cada par (x,y) del array 2D le corresponde el color de un pixel.

BMP FILE HEADER



Cabe destacar que el fichero BMP, además del buffer de datos, contiene una cabecera con metainformación. Es por ello la declaración de algunos tipos en el código mostrado a continuación.

```
/*
    Programa:     bitmap_gen_test.c
    Descripción:  Genera un imagen bitmap 512x512 en formato BMP y la guarda
en el fichero test.bmp.
    Compilación:  gcc --32 -o bitmap_gen_test bitmap_gen_test.c
    Ejecución:    $./bitmap_gen_test
    Visualización: comando "display" -> $display test.bmp -> salir con Ctrl-C
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Definición de tipos
typedef int LONG;                // 4 bytes (-2^31 a 2^31-1)
typedef unsigned char BYTE;      // 1 byte (0 a 2^8-1)
typedef unsigned int DWORD;      // 4 bytes (0 a 2^32-1)
typedef unsigned short WORD;     // 2 bytes (0 a 2^16-1)

typedef struct tagBITMAPFILEHEADER
{
    WORD    bfType;        // 2  /* Magic identifier */
    DWORD   bfSize;        // 4  /* File size in bytes */
    WORD    bfReserved1;   // 2
    WORD    bfReserved2;   // 2
    DWORD   bfOffBits;     // 4 /* Offset to image data, bytes */
} __attribute__((packed)) BITMAPFILEHEADER;

typedef struct tagBITMAPINFOHEADER
{
    DWORD    biSize;       // 4 /* Header size in bytes */
    LONG     biWidth;      // 4 /* Width of image */
    LONG     biHeight;     // 4 /* Height of image */
    WORD     biPlanes;     // 2 /* Number of colour planes */
    WORD     biBitCount;   // 2 /* Bits per pixel */
    DWORD    biCompress;   // 4 /* Compression type */
    DWORD    biSizeImage;  // 4 /* Image size in bytes */
    LONG     biXPelsPerMeter;    // 4
    LONG     biYPelsPerMeter;    // 4 /* Pixels per meter */
    DWORD    biClrUsed;          // 4 /* Number of colours */
    DWORD    biClrImportant;     // 4 /* Important colours */
} __attribute__((packed)) BITMAPINFOHEADER;

/*
    typedef struct tagRGBQUAD
    {
        unsigned char    rgbBlue;
```

```
        unsigned char    rgbGreen;
        unsigned char    rgbRed;
        unsigned char    rgbReserved;
    } RGBQUAD;

    for biBitCount is 16/24/32, it may be useless
*/

typedef struct
{
    BYTE    b;      // byte reservado para color blue (azul)
    BYTE    g;      // byte reservado para color green (verde)
    BYTE    r;      // byte reservado para color red (rojo)
} RGB_data;         // TIPO RGB -> buffer de tipo RGB_data

int bmp_generator(char *filename, int width, int height, unsigned char *data)
// parametros:
      // nombre fichero que quiero que me genere, altura, anchura, tabla 2D
{
    BITMAPFILEHEADER bmp_head;   // variable bmp_head de tipo BITMAPFILEHEADER
    BITMAPINFOHEADER bmp_info;   // variable bmp_info de tipo BITMAPINFOHEADER
    int size = width * height * 3; // tamaño = largo x ancho x 3 B/pixel

    bmp_head.bfType = 0x4D42;    // 'BM' -> 0x4D = "M" | 0x42 = "B"; BM=BitMap
    bmp_head.bfSize= size + sizeof(BITMAPFILEHEADER) +
sizeof(BITMAPINFOHEADER);        // tamaño data + tam.header + tam.metainfo
    bmp_head.bfReserved1 = bmp_head.bfReserved2 = 0;
    bmp_head.bfOffBits = bmp_head.bfSize - size;//byte offset to start of img
    // finish the initial of head

    bmp_info.biSize = 40;        // tamaño de cabecera
    bmp_info.biWidth = width;    // ancho (pixeles) de imagen
    bmp_info.biHeight = height;  // alto (pixeles) de imagen
    bmp_info.biPlanes = 1;
    bmp_info.biBitCount = 24;    // bits/pixel -> 3B/pixel; 1B=8b; 3x8=24bits
    bmp_info.biCompress = 0;  // compression type -> only 4&8 bit images comp.
    bmp_info.biSizeImage = size; // size of compressed file
    bmp_info.biXPelsPerMeter = 0;// Horizontal Resolution - pixels/meter
    bmp_info.biYPelsPerMeter = 0;// Vertical Resolution - pixels/meter
    bmp_info.biClrUsed = 0 ;     // colors in color table
    bmp_info.biClrImportant = 0; // Important color count
    // finish the initial of infohead;

    // copy the data
    FILE *fp;
    if (!(fp = fopen(filename, "wb"))) return 0;
      // abrir fichero en modo WriteBinary -> comprobando si da error

// escribir datos cabecera
    fwrite(&bmp_head, 1, sizeof(BITMAPFILEHEADER), fp);
// escribir meta informacion cabecera
```

```
    fwrite(&bmp_info, 1, sizeof(BITMAPINFOHEADER), fp);
// escribir la matriz data (buffer)
    fwrite(data, 1, size, fp);
    fclose(fp);     // cerrar fichero

    return 1;       // devolver 1
}

int main(int argc, char **argv)
{
    int i, j;       // variables para recorrer la matriz 2D

    RGB_data buffer[512][512]; // defino buffer, matriz de tipo RGB_data de
512x512

    memset(buffer, 0, sizeof(buffer));  // establezco todos los bytes de
buffer a 0 -> color negro de fondo

    for (i = 0; i < 256; i++)// recorremos la primera mitad a lo alto/ancho
    {
        for (j = 0; j < 256; j++) // recorremos la primera mitad a lo largo
        {
            buffer[i][j].g = buffer[i][j].b = 0x7f; // la combinacion es un
gris al 50%
            buffer[i][j].r = 0x7f;
        }
    }

    bmp_generator("./test.bmp", 512, 512, (BYTE*)buffer);
      // llamada a accion para generar imagen en fichero test.bmp de tamaño
512x512 con el contenido en matriz buffer

    return EXIT_SUCCESS;  // devolver al sistema operativo valor EXIT_SUCCESS
}
```

Tras compilar este programa se obtiene un fichero ejecutable (`bitmap_gen_test`) listo para ser cargado en memoria. Cuando se ejecuta y se carga en memoria, con el comando (`./bitmap_gen_test`), se genera otro fichero (test.bmp) que contiene la imagen en formato BMP generada por el propio programa. Para visualizar dicha imagen hacemos uso del comando `display` y el nombre del fichero a visualizar. En este caso utilizamos el comando `display test.bmp`, obteniendo la siguiente imagen:

SALIDA BITMAP_GEN_TEST.C

Inicialmente se define el tamaño de la imagen de 512x512. Después con la función `memset` se inicializa todo el buffer (matriz de 512x512) a 0.

```
memset(buffer, 0, sizeof(buffer));
```

Teniendo la siguiente string

```
str = "This is string.h library function"
```

Cuando se usa la función `memset()` lo que se consigue es lo siguiente.

```
memset(str, '$', 7);
```

Inicializamos los siete primeros bytes de la string `str`, con el signo dólar.

De tal forma que la string queda de la siguiente forma:

```
$$$$$$$ string.h library function
```

En nuestro caso, en vez de tener una string tenemos la matriz la cual tiene las proporciones de los colores, para más adelante construir la imagen. En vez de tener el signo dólar, tenemos el carácter 0, que representa la ausencia de color. Y, por último, en vez de los 7 primeros bytes de la string tenemos el número de bytes del buffer, gracias a la función `sizeof()`.

Esto lo que hace es inicializar todos y cada uno de los pixeles R-G-B a 0x00-0x00-0x00. Al no tener ningún color (falta/ausencia de color), es por ello que se inicializa con color negro.

A continuación, en el primer bucle `for` se usa una variable (i) para recorrer la primera mitad (0 a 255) y con el segundo bucle `for` lo que hacemos es recorrer la primera mitad en la otra dimensión. Es decir, para cada valor de i (diferentes alturas) recorremos la mitad de la imagen a lo largo.
De esta forma lo que conseguimos con estos bucles es recorrer la esquina inferior izquierda de dicha imagen, inicializada a color negro, para cambiar el color de dicho trozo. El nuevo color que se ha decidido es una combinación del Red (rojo), Green (verde) y Blue (azul) al 50%.

Al final del programa, el programa devuelve al sistema operativo el valor EXIT_SUCCESS para indicar que finaliza correctamente. Este valor se puede ver gracias al comando `echo $?` tras compilar y ejecutar el programa.

```
sayechu@sayechu-MacBookPro:~/Escritorio/COD2$ gcc -m32 -g -o bitmap_gen_test bitmap_gen_test.c
sayechu@sayechu-MacBookPro:~/Escritorio/COD2$ ./bitmap_gen_test
sayechu@sayechu-MacBookPro:~/Escritorio/COD2$ echo $?
0
sayechu@sayechu-MacBookPro:~/Escritorio/COD2$
```

cuadrado_128x128.c

```
/*
    Programa:     cuadrado_128x128.c
    Descripción:  Genera un imagen bitmap DIMENSIONxDIMENSION en formato BMP y
la guarda en el fichero test.bmp.
    Compilación:  gcc --32 -o cuadrado_128x128 cuadrado_128x128.c
    Ejecución:    $./cuadrado_128x128
    Visualización: comando "display" -> $display test.bmp -> salir con Ctrl-C
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// MACROS
#define DIMENSION 128      // definimos dimensión imagen -> 128x128

// Definición de tipos
typedef int LONG;                  // 4 bytes; rango valores (-2^31 a 2^31-1)
typedef unsigned char BYTE;        // 1 byte; rango valores (0 a 2^8-1)
typedef unsigned int DWORD;        // 4 bytes; rango valores (0 a 2^32-1)
typedef unsigned short WORD;       // 2 bytes; rango valores (0 a 2^16-1)

typedef struct tagBITMAPFILEHEADER      // tipo para cabecera
{
    WORD    bfType;                // 2  /* Magic identifier */
    DWORD   bfSize;                // 4  /* File size in bytes */
    WORD    bfReserved1;           // 2
    WORD    bfReserved2;           // 2
    DWORD   bfOffBits;             // 4 /* Offset to image data, bytes */
} __attribute__((packed)) BITMAPFILEHEADER;
```

```
typedef struct tagBITMAPINFOHEADER     // tipo con metainformación de cabecera
{
    DWORD    biSize;             // 4 /* Header size in bytes */
    LONG     biWidth;            // 4 /* Width of image */
    LONG     biHeight;           // 4 /* Height of image */
    WORD     biPlanes;           // 2 /* Number of colour planes */
    WORD     biBitCount;         // 2 /* Bits per pixel */
    DWORD    biCompress;         // 4 /* Compression type */
    DWORD    biSizeImage;        // 4 /* Image size in bytes */
    LONG     biXPelsPerMeter;    // 4
    LONG     biYPelsPerMeter;    // 4 /* Pixels per meter */
    DWORD    biClrUsed;          // 4 /* Number of colours */
    DWORD    biClrImportant;     // 4 /* Important colours */
} __attribute__((packed)) BITMAPINFOHEADER;

/*
    typedef struct tagRGBQUAD
    {
        unsigned char    rgbBlue;
        unsigned char    rgbGreen;
        unsigned char    rgbRed;
        unsigned char    rgbReserved;
    } RGBQUAD;

    for biBitCount is 16/24/32, it may be useless
*/

typedef struct
{
    BYTE    b;      // byte reservado para color blue (azul)
    BYTE    g;      // byte reservado para color green (verde)
    BYTE    r;      // byte reservado para color red (rojo)
} RGB_data;         // TIPO RGB -> buffer de tipo RGB_data

int bmp_generator(char *filename, int width, int height, unsigned char *data)
// parametros: nombre de fichero generado, altura, anchura, matriz 2D
{
    BITMAPFILEHEADER bmp_head;   // variable bmp_head de tipo BITMAPFILEHEADER
    BITMAPINFOHEADER bmp_info;   // variable bmp_info de tipo BITMAPINFOHEADER
    int size = width * height * 3; // tamaño = largo x ancho x 3 B/pixel

    bmp_head.bfType = 0x4D42;    // 'BM' -> 0x4D = "M" | 0x42 = "B"; BM=BitMap
    bmp_head.bfSize= size + sizeof(BITMAPFILEHEADER) +
sizeof(BITMAPINFOHEADER);        // tamaño data + tam.header + tam.metainfo
    bmp_head.bfReserved1 = bmp_head.bfReserved2 = 0;
    bmp_head.bfOffBits = bmp_head.bfSize - size;//byte offset to start of img
    // finish the initial of head

    bmp_info.biSize = 40;        // tamaño de cabecera
    bmp_info.biWidth = width;    // ancho (pixeles) de imagen
```

```
    bmp_info.biHeight = height;  // alto (pixeles) de imagen
    bmp_info.biPlanes = 1;
    bmp_info.biBitCount = 24;    // bits/pixel -> 3B/pixel; 1B=8b; 3x8=24bits
    bmp_info.biCompress = 0;  // compression type -> only 4&8 bit images comp.
    bmp_info.biSizeImage = size; // size of compressed file
    bmp_info.biXPelsPerMeter = 0;// Horizontal Resolution - pixels/meter
    bmp_info.biYPelsPerMeter = 0;// Vertical Resolution - pixels/meter
    bmp_info.biClrUsed = 0 ;     // colors in color table
    bmp_info.biClrImportant = 0; // Important color count
    // finish the initial of infohead;

    // copy the data
    FILE *fp;
    if (!(fp = fopen(filename, "wb"))) return 0;
      // abrir fichero en modo WriteBinary -> comprobando si da error

// escribir datos cabecera
    fwrite(&bmp_head, 1, sizeof(BITMAPFILEHEADER), fp);
// escribir meta informacion cabecera
    fwrite(&bmp_info, 1, sizeof(BITMAPINFOHEADER), fp);
// escribir la matriz data (buffer)
    fwrite(data, 1, size, fp);
    fclose(fp);     // cerrar fichero

    return 1;       // devolver 1
}

int main(int argc, char **argv)
{
    int i, j; // variables para recorrer la matriz 2D

    RGB_data buffer[DIMENSION][DIMENSION]; // defino buffer, matriz de tipo
RGB_data de DIMENSIONxDIMENSION = 128x128

    memset(buffer, 0, sizeof(buffer)); // establezco todos los bytes de buffer
a 0 -> color negro de fondo

    for (i = 0; i < DIMENSION/2; i++) // recorremos la primera mitad a lo
alto/ancho
    {
        for (j = 0; j < DIMENSION/2; j++) // recorremos la primera mitad a lo
largo para cada valor de i. -> para i=0; j de 0..63; para i=1; j de 0..63; …

        {
             buffer[i][j].g = 0x7f;     // color green-verde al 50%
             buffer[i][j].b = 0x7f;     // color blue-azul al 50%
             buffer[i][j].r = 0x7f;     // color red-rojo al 50%
        }    // COMBINACION DE VERDE,AZUL Y ROJO AL 50% -> GRIS
    }

    bmp_generator("./test.bmp", DIMENSION, DIMENSION, (BYTE*)buffer);
```

```
// llamada a accion para generar imagen en fichero test.bmp de tamaño 512x512
con el contenido en matriz buffer
    return EXIT_SUCCESS; // devolver al sistema operativo valor EXIT_SUCCESS
}
```

Este último programa es idéntico al primer programa (`bitmap_gen_test.c`) solo que el tamaño (largo y ancho de imagen que queremos generar) en vez de pasarlo a la acción (`bmp_generator`) como un entero, se ha declarado una MACRO, con las dimensiones de dicha imagen, y se le pasa el valor definido en esa MACRO.

En este caso hemos definido la macro `DIMENSION` con valor de 128. Esto es, nos va a generar una imagen de 128x128. Al igual que en el caso anterior, definimos dos bucles tales que i tiene rango de 0 a 63 (primera mitad a lo alto). Para cada valor de i, j toma los valores de 0 a 63. Con esto conseguimos recorrer la primera mitad inferior izquierda cambiando el color, inicializado a negro (ausencia de color), para cambiarlo a una combinación de Red (rojo), Green (verde) y Blue (azul) al 50%, esto es, un gris.

Tras compilar y ejecutar dicho programa, se obtiene la siguiente imagen:

SALIDA CUADRADO_128X128.C

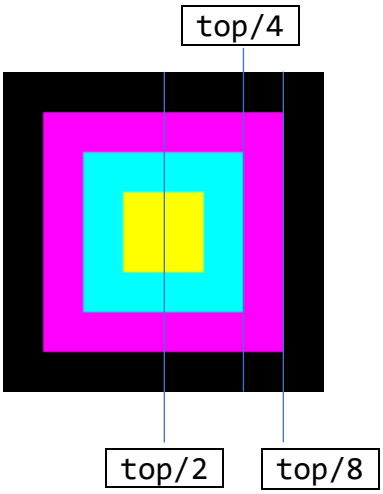
cuadrados_4.c

```
/*
    Programa:     cuadrados_4.c
    Descripción:  Genera cuatro rectangulos anidados bitmap en formato BMP y
guarda la imagen en el fichero test.bmp.
    Compilación:  gcc --32 -o cuadrados_4 cuadrados_4.c
    Ejecución:    $./cuadrados_4
    Visualización: comando "display" -> $display test.bmp -> salir con Ctrl-C
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define top   512          // DIMENSION IMAGEN -> 512X512
#define xcoor top/8        // MACRO PARA ACCEDER A CADA CUADRADO
#define ycoor top/8        // MACRO PARA ACCEDER A CADA CUADRADO

// Definición de tipos
typedef int LONG;                 // 4 bytes; rango valores (-2^31 a 2^31-1)
typedef unsigned char BYTE;       // 1 byte; rango valores (0 a 2^8-1)
```

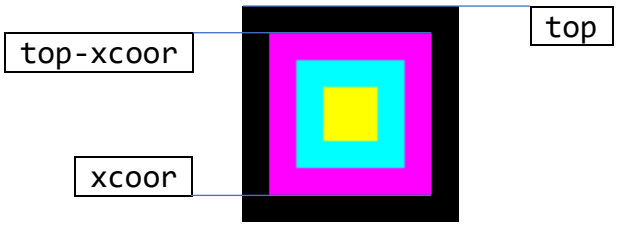
```
typedef unsigned int DWORD;      // 4 bytes; rango valores (0 a 2^32-1)
typedef unsigned short WORD;     // 2 bytes; rango valores (0 a 2^16-1)

// definición del tipo de la cabecera FILE del fichero BMP
typedef struct tagBITMAPFILEHEADER     // tipo para cabecera
{
    WORD    bfType;              // 2  /* Magic identifier */
    DWORD   bfSize;              // 4  /* File size in bytes */
    WORD    bfReserved1;         // 2
    WORD    bfReserved2;         // 2
    DWORD   bfOffBits;           // 4 /* Offset to image data, bytes */
} __attribute__((packed)) BITMAPFILEHEADER;

// definición del tipo de la cabecera INFO del fichero BMP
typedef struct tagBITMAPINFOHEADER     // tipo con metainformación de cabecera
{
    DWORD    biSize;             // 4 /* Header size in bytes */
    LONG     biWidth;            // 4 /* Width of image */
    LONG     biHeight;           // 4 /* Height of image */
    WORD     biPlanes;           // 2 /* Number of colour planes */
    WORD     biBitCount;         // 2 /* Bits per pixel */
    DWORD    biCompress;         // 4 /* Compression type */
    DWORD    biSizeImage;        // 4 /* Image size in bytes */
    LONG     biXPelsPerMeter;    // 4
    LONG     biYPelsPerMeter;    // 4 /* Pixels per meter */
    DWORD    biClrUsed;          // 4 /* Number of colours */
    DWORD    biClrImportant;     // 4 /* Important colours */
} __attribute__((packed)) BITMAPINFOHEADER;

// definición del tipo de cada pixel. Cada pixel son tres bytes . Cada byte es
tipo BYTE.
typedef struct
{
    BYTE    b;      // byte reservado para color blue (azul)
    BYTE    g;      // byte reservado para color green (verde)
    BYTE    r;      // byte reservado para color red (rojo)
} RGB_data;         // TIPO RGB -> buffer de tipo RGB_data


int bmp_generator(char *filename, int width, int height, unsigned char *data)
{
    BITMAPFILEHEADER bmp_head;   // variable bmp_head de tipo BITMAPFILEHEADER
    BITMAPINFOHEADER bmp_info;   // variable bmp_info de tipo BITMAPINFOHEADER
    int size = width * height * 3; // tamaño = largo x ancho x 3 B/pixel

    bmp_head.bfType = 0x4D42;    // 'BM' -> 0x4D = "M" | 0x42 = "B"; BM=BitMap
    bmp_head.bfSize= size + sizeof(BITMAPFILEHEADER) +
sizeof(BITMAPINFOHEADER);        // tamaño data + tam.header + tam.metainfo
    bmp_head.bfReserved1 = bmp_head.bfReserved2 = 0;
    bmp_head.bfOffBits = bmp_head.bfSize - size;//byte offset to start of img
    // finish the initial of head
```

```
    bmp_info.biSize = 40;        // tamaño de cabecera
    bmp_info.biWidth = width;    // ancho (pixeles) de imagen
    bmp_info.biHeight = height;  // alto (pixeles) de imagen
    bmp_info.biPlanes = 1;
    bmp_info.biBitCount = 24;    // bits/pixel -> 3B/pixel; 1B=8b; 3x8=24bits
    bmp_info.biCompress = 0;  // compression type -> only 4&8 bit images comp.
    bmp_info.biSizeImage = size; // size of compressed file
    bmp_info.biXPelsPerMeter = 0;// Horizontal Resolution - pixels/meter
    bmp_info.biYPelsPerMeter = 0;// Vertical Resolution - pixels/meter
    bmp_info.biClrUsed = 0 ;     // colors in color table
    bmp_info.biClrImportant = 0; // Important color count
    // finish the initial of infohead;

    // copy the data
    FILE *fp;
    if (!(fp = fopen(filename, "wb"))) return 0;
      // abrir fichero en modo WriteBinary -> comprobando si da error

// escribir datos cabecera
    fwrite(&bmp_head, 1, sizeof(BITMAPFILEHEADER), fp);
// escribir meta informacion cabecera
    fwrite(&bmp_info, 1, sizeof(BITMAPINFOHEADER), fp);
// escribir la matriz data (buffer)
    fwrite(data, 1, size, fp);
    fclose(fp);     // cerrar fichero

    return 1;       // devolver 1
}

// Función de entrada
int main(int argc, char **argv)
{
    int i,j;

    RGB_data buffer[top][top]; // Array de pixels

    memset(buffer, 0, sizeof(buffer)); //  inicializa el buffer con el valor
cero. Librería libc.
    // Los colores se expresan en formato RGB donde la intensidad de cada
color se codifica con un byte
    // 0x00: ausencia de color ; 0xFF: intensidad máxima
    // La ausencia de los tres colores primarios R=G=B=0x00 es el negro
    // R=G=B=0xFF es el blanco

    for (i = xcoor; ((xcoor <= i) && (i < (top-xcoor))); i++)
    {
        for (j = ycoor; ((ycoor <= j) && (j < (top-ycoor))); j++)
        {
        // SE COMBINA ROJO Y AZUL -> ROSA
        // intensidad de rojo
```

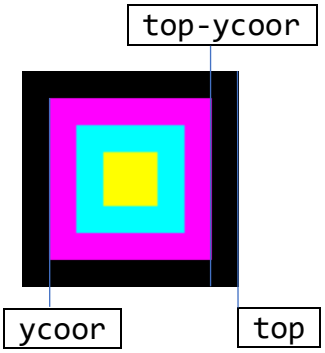
```
            buffer[i][j].r = 0xff;      // rojo al 100%
        // intensidad de verde
            buffer[i][j].g = 0x00;
        // intensidad de azul
            buffer[i][j].b = 0xff;      // azul al 100%
        }
    }

for (i = 2*xcoor; ((2*xcoor <= i) && (i < (top-2*xcoor))); i++)
    {
        for (j = 2*ycoor; ((2*ycoor <= j) && (j < (top-2*ycoor))); j++)
        {
        // SE COMBINA VERDE Y AZUL -> CIAN
        // intensidad de rojo
            buffer[i][j].r = 0x00;
        // intensidad de verde
            buffer[i][j].g = 0xff;      // verde al 100%
        // intensidad de azul
            buffer[i][j].b = 0xff;      // azul al 100%
        }
    }

for (i = 3*xcoor; ((3*xcoor <= i) && (i < (top-3*xcoor))); i++)
    {
        for (j = 3*ycoor; ((3*ycoor <= j) && (j < (top-3*ycoor))); j++)
        {
        // SE COMBINA ROJO Y VERDE -> AMARILLO
        // intensidad de rojo
            buffer[i][j].r = 0xFF;      // rojo al 100%
        // intensidad de verde
            buffer[i][j].g = 0xff;      // verde al 100%
        // intensidad de azul
            buffer[i][j].b = 0x00;
        }
    }

    bmp_generator("./test.bmp", top, top, (BYTE*)buffer);

    return EXIT_SUCCESS;
}
```



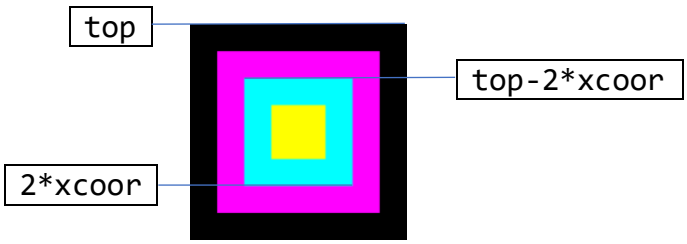

```
for (i = xcoor; ((xcoor <= i) && (i < (top-xcoor))); i++)
```



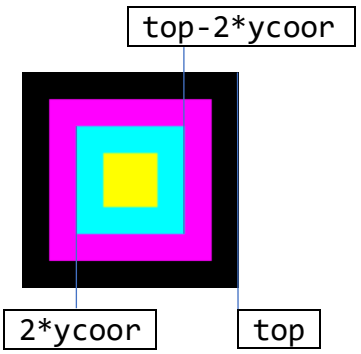

```
for (j = ycoor; ((ycoor <= j) && (j < (top-ycoor))); j++)
```



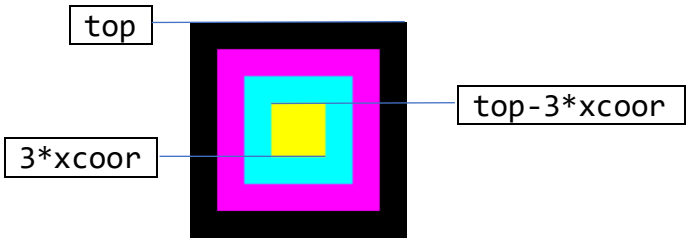

```
for (i = 2*xcoor; ((2*xcoor <= i) && (i < (top-2*xcoor))); i++)
```



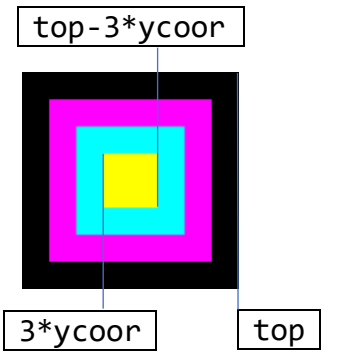
```
for (j = 2*ycoor; ((2*ycoor <= j) && (j < (top-2*ycoor))); j++)
```




```
for (i = 3*xcoor; ((3*xcoor <= i) && (i < (top-3*xcoor))); i++)
```

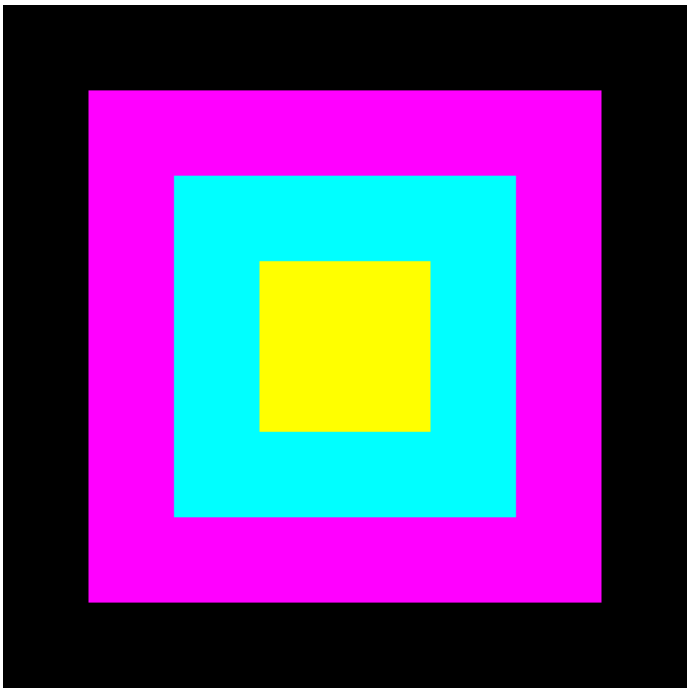



```
for (j = 3*ycoor; ((3*ycoor <= j) && (j < (top-3*ycoor))); j++)
```



Vistos los rangos de cada uno de los bucles for y coloreado los 4 cuadrados (negro, rosa, cian y amarillo), se muestra la imagen generada en tamaño 512x512 a continuación:

SALIDA CUADRADOS_4.C



bmp_funcion.c

```
/*
    Programa:       bmp_funcion.c
    Descripción:    Genera un rectángulo en una imagen bitmap en formato BMP y
la guarda en el fichero test.bmp.
    Funciones:      void pixels_generator(unsigned int x, unsigned int y,
unsigned int maximo, RGB_data reg_mem[][top])
            Argumentos:
                x -> origen de coordenadas del primer pixel del bucle para las
filas
                y -> origen de coordenadas del primer pixel del bucle para las
columnas
                maximo -> máxima coordenada del cuadrado en el bucle para
filas y columnas
                reg_mem -> array 2D de pixeles de tipo RGB_data
    Compilación:  gcc --32 -o bmp_funcion bmp_funcion.c
    Ejecución:    $./bmp_funcion
    Visualización: comando "display" -> $display test.bmp -> salir con Ctrl-C
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define top   512          // DIMENSION IMAGEN -> 512X512
#define xcoor top/8        // MACRO PARA ACCEDER A CADA CUADRADO
#define ycoor top/8        // MACRO PARA ACCEDER A CADA CUADRADO
```

```
// Definición de tipos
typedef int LONG;                // 4 bytes; rango valores (-2^31 a 2^31-1)
typedef unsigned char BYTE;      // 1 byte; rango valores (0 a 2^8-1)
typedef unsigned int DWORD;      // 4 bytes; rango valores (0 a 2^32-1)
typedef unsigned short WORD;     // 2 bytes; rango valores (0 a 2^16-1)

// definición del tipo de la cabecera FILE del fichero BMP
typedef struct tagBITMAPFILEHEADER
{
    WORD    bfType;              // 2  /* Magic identifier */
    DWORD   bfSize;              // 4  /* File size in bytes */
    WORD    bfReserved1;         // 2
    WORD    bfReserved2;         // 2
    DWORD   bfOffBits;           // 4 /* Offset to image data, bytes */
} __attribute__((packed)) BITMAPFILEHEADER;

// definición del tipo de la cabecera INFO del fichero BMP
typedef struct tagBITMAPINFOHEADER
{
    DWORD    biSize;             // 4 /* Header size in bytes */
    LONG     biWidth;            // 4 /* Width of image */
    LONG     biHeight;           // 4 /* Height of image */
    WORD     biPlanes;           // 2 /* Number of colour planes */
    WORD     biBitCount;         // 2 /* Bits per pixel */
    DWORD    biCompress;         // 4 /* Compression type */
    DWORD    biSizeImage;        // 4 /* Image size in bytes */
    LONG     biXPelsPerMeter;    // 4
    LONG     biYPelsPerMeter;    // 4 /* Pixels per meter */
    DWORD    biClrUsed;          // 4 /* Number of colours */
    DWORD    biClrImportant;     // 4 /* Important colours */
} __attribute__((packed)) BITMAPINFOHEADER;


// definición del tipo de cada pixel. Cada pixel son tres bytes. Cada byte es
tipo BYTE.
typedef struct
{
    BYTE    b;      // byte reservado para color blue (azul)
    BYTE    g;      // byte reservado para color green (verde)
    BYTE    r;      // byte reservado para color red (rojo)
} RGB_data;         // TIPO RGB -> buffer de tipo RGB_data


int bmp_generator(char *filename, int width, int height, BYTE *data)
{
    BITMAPFILEHEADER bmp_head;   // variable bmp_head de tipo BITMAPFILEHEADER
    BITMAPINFOHEADER bmp_info;   // variable bmp_info de tipo BITMAPINFOHEADER
    int size = width * height * 3; // tamaño = largo x ancho x 3 B/pixel

    bmp_head.bfType = 0x4D42;    // 'BM' -> 0x4D = "M" | 0x42 = "B"; BM=BitMap
```

```
    bmp_head.bfSize= size + sizeof(BITMAPFILEHEADER) +
sizeof(BITMAPINFOHEADER);        // tamaño data + tam.header + tam.metainfo
    bmp_head.bfReserved1 = bmp_head.bfReserved2 = 0;
    bmp_head.bfOffBits = bmp_head.bfSize - size;//byte offset to start of img
    // finish the initial of head

    bmp_info.biSize = 40;        // tamaño de cabecera
    bmp_info.biWidth = width;    // ancho (pixeles) de imagen
    bmp_info.biHeight = height;  // alto (pixeles) de imagen
    bmp_info.biPlanes = 1;
    bmp_info.biBitCount = 24;    // bits/pixel -> 3B/pixel; 1B=8b; 3x8=24bits
    bmp_info.biCompress = 0;  // compression type -> only 4&8 bit images comp.
    bmp_info.biSizeImage = size; // size of compressed file
    bmp_info.biXPelsPerMeter = 0;// Horizontal Resolution - pixels/meter
    bmp_info.biYPelsPerMeter = 0;// Vertical Resolution - pixels/meter
    bmp_info.biClrUsed = 0 ;     // colors in color table
    bmp_info.biClrImportant = 0; // Important color count
    // finish the initial of infohead;

    // copy the data
    FILE *fp;
    if (!(fp = fopen(filename, "wb"))) return 0;
      // abrir fichero en modo WriteBinary -> comprobando si da error

// escribir datos cabecera
    fwrite(&bmp_head, 1, sizeof(BITMAPFILEHEADER), fp);
// escribir meta informacion cabecera
    fwrite(&bmp_info, 1, sizeof(BITMAPINFOHEADER), fp);
// escribir la matriz data (buffer)
    fwrite(data, 1, size, fp);
    fclose(fp);     // cerrar fichero

    return 1;       // devolver 1
}

void pixels_generator(unsigned int x, unsigned int y, unsigned int maximo,
RGB_data reg_mem[][top])
{
    int i, j;

    for (i = x; ((x <= i) && (i < (top - x))); i++)
    {
        for (j = y; ((y <= j) && (j < (top - y))); j++)
        {
        // combinación de ROJO y AZUL al 100%
        // intensidad de rojo
            reg_mem[i][j].r = 0xff;     // rojo al 100%
        // intensidad de verde
            reg_mem[i][j].g = 0x00;
        // intensidad de azul
            reg_mem[i][j].b = 0xff;     // azul al 100%
```

```
        }
    }

}
// Función de entrada
int main(int argc, char **argv)
{
    RGB_data buffer[top][top]; // Array de pixels

    memset(buffer, 0, sizeof(buffer)); //  inicializa el buffer con el valor
cero. Librería libc.
    // Los colores se expresan en formato RGB donde la intensidad de cada
color se codifica con un byte
    // 0x00: ausencia de color ; 0xFF: intensidad máxima
    // La ausencia de los tres colores primarios R=G=B=0x00 es el negro
    // R=G=B=0xFF es el blanco

    pixels_generator(xcoor,ycoor,top,buffer);

    bmp_generator("./test.bmp", top, top, (BYTE*)buffer); // casting a buffer.
Definido como RGB_data y pasado como BYTE

    return EXIT_SUCCESS;
}
```

Este programa es idéntico a los ya vistos anteriormente solo que en vez de hacer los bucles de la matriz buffer en el programa principal main, se ha creado una función que hace lo mismo. Esta función es llamada desde el programa principal main y, posteriormente, se genera la imagen llamando a la función `bmp_generator` con el contenido de la matriz.

En este caso, la función `pixels_generator` es la encargada de hacer el recorrido en la matriz cambiando el color. En esta función se han definido dos bucles para "dibujar" un cuadrado de color rosa (combinación de Red al 100%, Green al 0% y Blue al 100%). El rango de los bucles son los siguientes:

```
for (i = x; ((x <= i) && (i < (top - x))); i++)
```

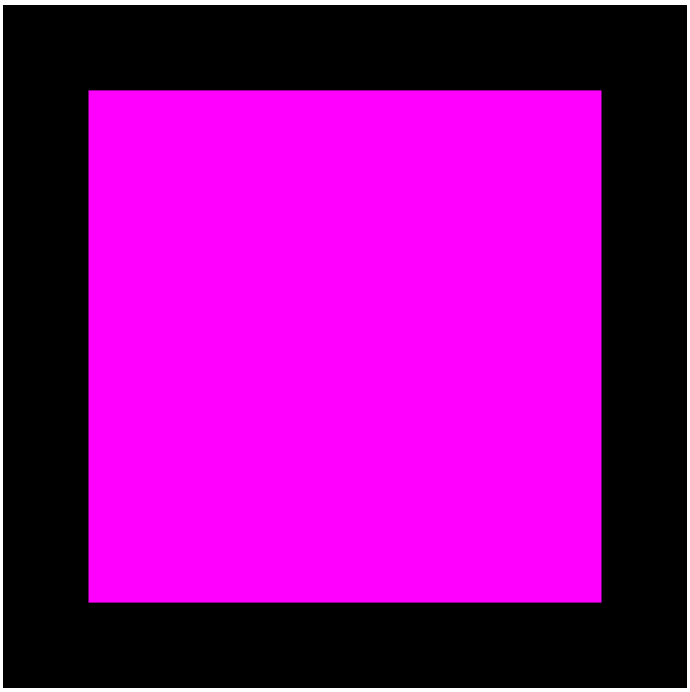
En este primer bucle, el rango es de `x` a `top-x-1`, donde `x = (top/8)`. Esto es, va desde `(top/8)` hasta `top-(top/8)-1`

```
for (j = y; ((y <= j) && (j < (top - y))); j++)
```

En este segundo bucle, anidado al primero, el rango es idéntico al anterior solo que en la otra dimensión. Esto es, va desde `y` hasta `top-y-1`, donde `y = (top/8)`.

Después de compilar y ejecutar el programa se ha obtenido la siguiente imagen.

SALIDA CUADRADOS_4.C



Programación en ASM

bmp_as.c

```
/*
    Programa:       bmp_as.c
    Descripción:    Genera un rectángulo en una imagen bitmap en formato BMP y
la guarda en el fichero test_2a.bmp.
    Funciones:      void pixels_generator_2(unsigned int origen_x, unsigned
int origen_y, unsigned int lado, unsigned int proporcion, unsigned int
dimension, RGB_data reg_mem[][DIMENSION]);
            Argumentos:
                x -> origen de coordenadas del primer pixel del bucle para las
filas
                y -> origen de coordenadas del primer pixel del bucle para las
columnas
                lado ->
                proporcion ->
                dimension -> máxima coordenada del cuadrado en el bucle para
filas y columnas. No hace falta si ponemos maximo como top-x
                reg_mem -> array 2D de pixeles de tipo RGB_data
    Compilación:  gcc -m32 -o bmp_as bmp_as.c
    Ejecución:    $./bmp_as
    Visualización: comando "display" -> $display test_as.bmp -> salir con
Ctrl-C
*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define DIMENSION 1024    // Dimensión Imagen = 1024x1024
#define ORIGEN_X 0        // Macro que marca origen
#define ORIGEN_Y 0        // Macro que marca origen
#define LADO 512          // Tamaño de cuadrado a dibujar 512x512
#define PROPORCION 0x7f   // Rango: 0x00 - 0xFF -> 0x7f = 50%

// Definición de tipos
typedef int LONG;                // 4 bytes; rango valores (-2^31 a 2^31-1)
typedef unsigned char BYTE;      // 1 byte; rango valores (0 a 2^8-1)
typedef unsigned int DWORD;      // 4 bytes; rango valores (0 a 2^32-1)
typedef unsigned short WORD;     // 2 bytes; rango valores (0 a 2^16-1)

// definición del tipo de la cabecera FILE del fichero BMP
typedef struct tagBITMAPFILEHEADER
{
    WORD    bfType;              // 2  /* Magic identifier */
    DWORD   bfSize;              // 4  /* File size in bytes */
    WORD    bfReserved1;         // 2
    WORD    bfReserved2;         // 2
    DWORD   bfOffBits;           // 4 /* Offset to image data, bytes */
} __attribute__((packed)) BITMAPFILEHEADER;

// definición del tipo de la cabecera INFO del fichero BMP
typedef struct tagBITMAPINFOHEADER
{
    DWORD    biSize;             // 4 /* Header size in bytes */
    LONG     biWidth;            // 4 /* Width of image */
    LONG     biHeight;           // 4 /* Height of image */
    WORD     biPlanes;           // 2 /* Number of colour planes */
    WORD     biBitCount;         // 2 /* Bits per pixel */
    DWORD    biCompress;         // 4 /* Compression type */
    DWORD    biSizeImage;        // 4 /* Image size in bytes */
    LONG     biXPelsPerMeter;    // 4
    LONG     biYPelsPerMeter;    // 4 /* Pixels per meter */
    DWORD    biClrUsed;          // 4 /* Number of colours */
    DWORD    biClrImportant;     // 4 /* Important colours */
} __attribute__((packed)) BITMAPINFOHEADER;


// definición del tipo de cada pixel. Cada pixel son tres bytes. Cada byte es
un color, tipo BYTE.
typedef struct
{
    BYTE    b;      // byte reservado para color blue (azul)
    BYTE    g;      // byte reservado para color green (verde)
    BYTE    r;      // byte reservado para color red (rojo)
} RGB_data;         // TIPO RGB -> buffer de tipo RGB_data
```

```
int bmp_generator(char *filename, int width, int height, unsigned char *data)
// parametros: nombre de fichero generado, altura, anchura, matriz 2D
{
    BITMAPFILEHEADER bmp_head;   // variable bmp_head de tipo BITMAPFILEHEADER
    BITMAPINFOHEADER bmp_info;   // variable bmp_info de tipo BITMAPINFOHEADER
    int size = width * height * 3; // tamaño = largo x ancho x 3 B/pixel

    bmp_head.bfType = 0x4D42;    // 'BM' -> 0x4D = "M" | 0x42 = "B"; BM=BitMap
    bmp_head.bfSize= size + sizeof(BITMAPFILEHEADER) +
sizeof(BITMAPINFOHEADER);        // tamaño data + tam.header + tam.metainfo
    bmp_head.bfReserved1 = bmp_head.bfReserved2 = 0;
    bmp_head.bfOffBits = bmp_head.bfSize - size;//byte offset to start of img
    // finish the initial of head

    bmp_info.biSize = 40;        // tamaño de cabecera
    bmp_info.biWidth = width;    // ancho (pixeles) de imagen
    bmp_info.biHeight = height;  // alto (pixeles) de imagen
    bmp_info.biPlanes = 1;
    bmp_info.biBitCount = 24;    // bits/pixel -> 3B/pixel; 1B=8b; 3x8=24bits
    bmp_info.biCompress = 0;  // compression type -> only 4&8 bit images comp.
    bmp_info.biSizeImage = size; // size of compressed file
    bmp_info.biXPelsPerMeter = 0;// Horizontal Resolution - pixels/meter
    bmp_info.biYPelsPerMeter = 0;// Vertical Resolution - pixels/meter
    bmp_info.biClrUsed = 0 ;     // colors in color table
    bmp_info.biClrImportant = 0; // Important color count
    // finish the initial of infohead;

    // copy the data
    FILE *fp;
    if (!(fp = fopen(filename, "wb"))) return 0;
      // abrir fichero en modo WriteBinary -> comprobando si da error

// escribir datos cabecera
    fwrite(&bmp_head, 1, sizeof(BITMAPFILEHEADER), fp);
// escribir meta informacion cabecera
    fwrite(&bmp_info, 1, sizeof(BITMAPINFOHEADER), fp);
// escribir la matriz data (buffer)
    fwrite(data, 1, size, fp);
    fclose(fp);     // cerrar fichero

    return 1;       // devolver 1
}

void pixels_generator_2(unsigned int origen_x, unsigned int origen_y, unsigned
int lado, unsigned int proporcion, unsigned int dimension, RGB_data
reg_mem[][DIMENSION]);

// Función de entrada
int main(int argc, char **argv)
```

```
{
    RGB_data buffer[DIMENSION][DIMENSION]; // Matriz de pixeles

    memset(buffer, 0, sizeof(buffer)); //  inicializa el buffer con el valor
cero. Librería libc.
    // Los colores se expresan en formato RGB donde la intensidad de cada
color se codifica con un byte
    // 0x00: ausencia de color ; 0xFF: intensidad máxima
    // La ausencia de los tres colores primarios R=G=B=0x00 es el negro
    // R=G=B=0xFF es el blanco

    pixels_generator_2(ORIGEN_X, ORIGEN_Y, LADO, PROPORCION, DIMENSION,
buffer);

    bmp_generator("./test_as.bmp", DIMENSION, DIMENSION, (BYTE*)buffer); //
casting a buffer. Definido como RGB_data y pasado como BYTE

    exit(1);
}
```

Este programa es idéntico al bmp_funcion.c solo que tiene una particularidad respecto al mismo. La función encargada de "dibujar" la imagen, la que cambia el color del cuadrado dibujado, está programada en lenguaje ensamblador AT&T x86-32. Para ello se ha generado un fichero con la implementación de dicha función (pixels_generator_2) que se muestra el código a continuación:

Cabe recordar que para la compilación de este programa (bmp_as.c) será necesario incluir al módulo objeto pixels. Los comandos de compilación se encuentran en la su sección correspondiente de esta memoria.

pixels.s

```
/*
    Programa:    array_pixels_2.s
    Compilación:    as --32 -gstabs -o array_pixels_2.o array_pixels_2.s
    Descripción:     Generar un array de 2D (coordenadas de los pixels ) para
almacenar una estructura de datos RGB (3ª dimensión) e inicializarlo.
    Metodogía:   Desarrollar el algoritmo en lenguaje C
        Relacionar las estructuras de C con las de Ensamblador:
        Primero generamos la segunda dimensión
        Después generamos la primera dimensión con un array de direcciones de
memoria sin inicializar:
        A continuación inicializamos el array de punteros "  " con las
direcciones de los pixels
*/
    ## variables globales
    ## .global buffer       ## buffer es o una variable global o un argumento

    ## Sección Macros
    .equ DIMENSION_Z, 3     #tamaño de la tercera dimensión: estructura pixel
```

```
    ## El origen de coordenadas está en la esquina inferior izquierda.
    ## El contenido del buffer comienza por el origen de coordenadas y
contiene filas secuenciales
    ## La estructura del pixel sigue la seguencia B-G-R

    ## Sección variables scope fichero inicializadas
    .section .data
rojo:   .byte 0           ## (R)ed     - rojo
verde:  .byte 0           ## (G)reen   - verde
azul:   .byte 0           ## (B)lue    - azul
origen_x:   .4byte  0     ## Origen coordenada X del cuadrado en pixels
origen_y:   .4byte  0     ## Origen coordenada Y del cuadrado en pisels
lado:       .4byte  0     ## Lado del cuadrado en pixels
proporcion: .4byte  0     ## tanto por cien de los tres colores primarios RGB
dimension:  .4byte  0     ## Dimension del cuadrado background en pixels
buffer_ptr: .4byte  0     ## variable puntero donde guardaré el argumento
referencia que le pasa main desde C
fin_x:      .4byte  0     ## Limite coordenada X del cuadrado en
pixels=origen_x+lado-1
fin_y:      .4byte  0     ## Limite coordenada Y del cuadrado en
pixels=origen_y+lado-1


    ## Sección Instrucciones
    .section .text
    .global pixels_generator_2
    ##  .global buffer
    .type pixels_generator_2, @function
pixels_generator_2:
/*
    Correspondencia tuple 3D (Fila x_coor ,Columna y_coor, Offset RGB) ->
buffer lineal
    Coordenada Filas    :(0,Nx-1)
    Coordenada Columnas :(0,Ny-1)
    Offset RGB (0,1,2)
    Buffer (direcciona bytes):
    Elemento del buffer: cada elemento son 3 bytes.
    Espacio de elementos - Espacio bytes: la dirección de un elemento es el
byte AZUL de dicho elemento.
    1º elemento de cada fila i    : 1Fi -> x_coor * DIMENSION_Y * DIMENSION_Z
    1º elemento de cada columna j : 1Cj -> y_coor * DIMENSION_Z
    Posición del color:         PC (B->0, G->1, R->2)
    Función correspondencia del pixel (i,j) con el elemento (Offset Pixel
(i,j)) en el buffer: OPij -> 1Fi + 1Cj
    Ejemplo: Array XY  512x512, al pixel (3,3) le corresponde el siguiente
Offset:
        1F=3*512*3=4608
        1C=3*3    =9
        OP= 4608+9=4617
*/
```

```
    ## Nuevo frame en la pila
    push %ebp       ## apilar direccion a la que apunta EBP
    mov %esp, %ebp  ## EBP <- ESP; nuevo EBP = ESP
################################################
### SALVO EL CONTEXTO ANTERIOR A LA SUBRUTINA ###
    push %eax       ## apilar contenido registro EAX
    push %ebx       ## apilar contenido registro EBX
    push %ecx       ## apilar contenido registro ECX
    push %edx       ## apilar contenido registro EDX
    push %edi       ## apilar contenido registro EDI
    push %esi       ## apilar contenido registro ESI
################################################
## Cuando se hace la llamada desde programa C a la función pixels_generator_2
## se apilan los parámetros en orden inverso. Quedan así en la pila:
## pixels_generator_2(ORIGEN_X,ORIGEN_Y,LADO,PROPORCION,DIMENSION,buffer);
## En pila (de arriba abajo): ORIGEN_X, ORIGEN_Y, LADO, PROPORCION, DIMENSION,
## buffer
    ## Capturo el argumento origen_x SALTAR EBP apilado y RET ADDRESS, EBP+4+4
    mov 8(%ebp), %eax     # mover contenido en dirección:la que apunta EBP+8
    mov %eax, origen_x    # mover argumento captura a variable origen_x
    ## Capturo el argumento origen_y
    mov 12(%ebp), %eax    # mover contenido en dirección:la que apunta EBP+12
    mov %eax, origen_y    # mover argumento capturado a variable origen_y
    ## Capturo el argumento lado
    mov 16(%ebp),%eax     # mover contenido en dir a la que apunta EBP+16
    mov %eax,lado         # mover argumento capturado a variable lado
    ## Capturo el argumento proporcion
    mov 20(%ebp),%eax     # mover contenido en dir a la que apunta EBP+20
    mov %eax,proporcion   # mover argumento capturado a variable proporcion
    ## Capturo el argumento dimension
    mov 24(%ebp),%eax     # mover contenido en dir a la que apunta EBP+24
    mov %eax,dimension    # mover argumento capturado a variable dimension
    ## Capturo el argumento buffer
    mov 28(%ebp),%eax     # mover contenido en dir a la que apunta EBP+28
    mov %eax,buffer_ptr   # inicializo el puntero con el argumento

    ## intensidad colores: color INICIAL
    ## color background ubuntu RGB is (48, 10, 36).
    movb proporcion,%al
    movb $0x00,rojo       ## Red (rojo) al 0%
    movb $0xff,verde      ## Green (verde) al 100%
    movb $0x00,azul       ## Blue (azul) al 0$

    ## Cálculo de los limites    FIN_X = ORIGEN_x + LADO - 1
    mov origen_x,%eax     ## mover origen_x a registro EAX; EAX<-origen_x
    add lado,%eax         ## sumar EAX + lado; EAX <- EAX + lado
    dec %eax              ## decrementar EAX; EAX <- EAX - 1;
    mov %eax,fin_x        ## FIN_X <- EAX

    ## FIN_Y = ORIGEN_y + LADO - 1
    mov origen_y,%eax     ## mover origen_y a registro EAX; EAX<-origen_y
```

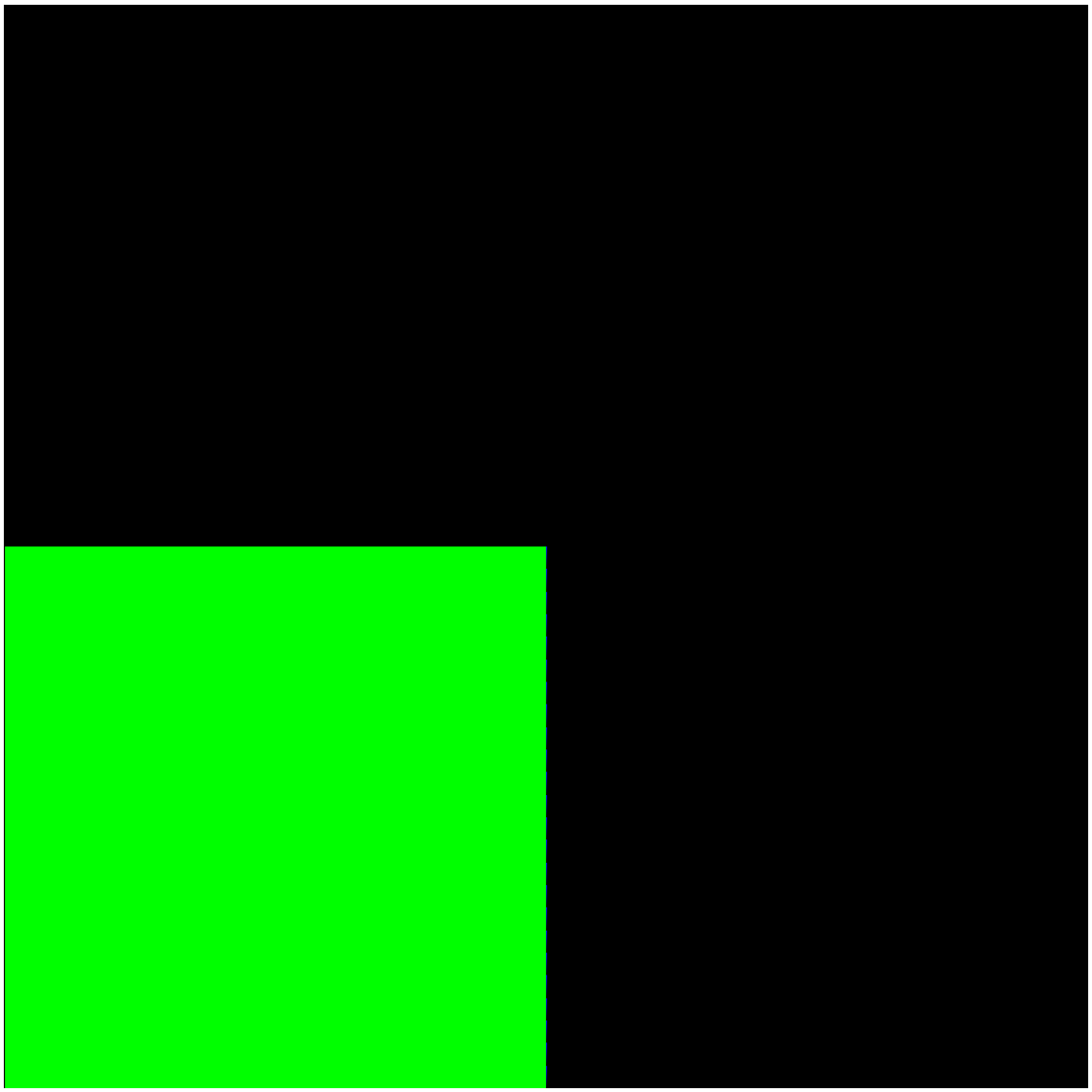
```
    add lado,%eax          ## sumar EAX + lado; EAX <- EAX + lado
    dec %eax               ## decrementar EAX; EAX <- EAX - 1;
    mov %eax,fin_y         ## FIN_Y <- EAX

    ## Bucle de inicialización del array
    mov origen_x,%edi       #inicio Filas; EDI registro índice con origen_x
fila:
    mov origen_y,%esi       #inicio Columnas; ESI registro índice con origen_y
columna:


        ##     control columna
    cmp fin_y, %esi        # comparamos origen_y con fin_y
    jz  col_exit           # si origen_y - fin_y = 0 -> salto a col_exit

    ## actualización posición columna
    inc %esi               # se incrementa origen_y, EMPIEZA EN 1 no en 0!!!
    ## Reset registros aritmética
    xor %eax,%eax          ## reset registro EAX; equivale a sub %eax, %eax
    xor %ebx,%ebx          ## reset registro EBX; equivale a sub %ebx, %ebx
    xor %ecx,%ecx          ## reset registro ECX; equivale a sub %ecx, %ecx
    ## Aritmética Fila  EQUIVALE A 3 x N x i -> posicion fila
    movw %di,%bx           # posición fila
    imul dimension,%ebx    # ebx <- ebx * dimension
    imul $DIMENSION_Z,%ebx# ebx <- ebx * DIMENSION_Z
    ## Aritmética Columna # posición columna
    movw %si,%cx           # cx <- si
    imul $DIMENSION_Z,%ecx# ecx <- ecx * DIMENSION_Z
    ## Correspondencia array_pixel -> posición buffer
    xor %eax,%eax   # reset registro EAX
    add %ebx,%eax   # EAX <- EAX + EBX
    add %ecx,%eax   # EAX <- EAX + ECX
    mov %eax,%ebx   # EBX contiene el offset del pixel en el buffer

    ## Actualizar colores en el pixel
    xor %ecx,%ecx       # índice color; Reset registro ECX; AZUL = 0
    movb azul,%al       # intensidad azul
    mov buffer_ptr,%edx # edx <- buffer_ptr
    lea (%edx,%ebx),%edx# edx <- Effective Address (%edx, %ebx)
    mov %eax,(%edx,%ecx) # dirección efectiva = M[buffer_ptr] + offset +
posi_color
    inc %ecx               # incrementar ECX; ECX <- ECX + 1; VERDE = 1
    movb verde,%al         # intensidad verde
    mov buffer_ptr,%edx    # edx <- buffer_ptr
    lea (%edx,%ebx),%edx   # edx <- Effective Address (%edx, %ebx)
    mov %eax,(%edx,%ecx) # dirección efectiva = M[buffer_ptr] + offset +
posi_color
    inc %ecx               # incrementar ECX; ECX <- ECX + 1; ROJO = 2
    movb rojo,%al          # intensidad rojo
    mov buffer_ptr,%edx    # edx <- buffer_ptr
    lea (%edx,%ebx),%edx   # edx <- M[buffer_ptr] + offset
```

```
    mov %eax,(%edx,%ecx) # dirección efectiva = M[buffer_ptr] + offset +
posi_color

    ## siguiente columna

    jmp columna           # salto a columna

    ## actualización posición fila
col_exit:
    cmp fin_x,%edi        # comparar fin_x con origen_x;
    jz fil_exit           # si origen_x-fin_x = 0; SALTO A fil_exit
    ##siguiente fila
    inc %edi              # edi <- edi + 1; origen_x

    ## Reset registros aritmética
    xor %eax,%eax         # reset registro eax
    xor %ebx,%ebx         # reset registro ebx
    xor %ecx,%ecx         # reset registro ecx
    ## Aritmética Fila
    movw %di,%bx          # posición fila
    imul dimension,%ebx   # ebx <- ebx*dimension
    imul $DIMENSION_Z,%ebx# ebx <- ebx * DIMENSION_Z
    ## Aritmética Columna
    movw %si,%cx          # posición columna
    imul $DIMENSION_Z,%ecx# ecx <- ecx * dimensión_z
    ## Correspondencia array_pixel -> posición buffer
    xor %eax,%eax         # reset registro eax
    add %ebx,%eax         # eax <- eax + ebx
    add %ecx,%eax         # eax <- eax + ecx
    mov %eax,%ebx         # EBX contiene el offset del pixel en el buffer

    ## Actualizar colores en el pixel
    xor %ecx,%ecx         # índice color
    movb azul,%al         # intensidad azul = 0
    mov buffer_ptr,%edx   # edx <- buffer_ptr
    lea (%edx,%ebx),%edx  # edx <- M[buffer_ptr] + offset
    mov %eax,(%edx,%ecx)  # dirección efectiva = M[buffer_ptr] + offset +
posi_color
    inc %ecx
    movb verde,%al        # intensidad verde = 1
    mov buffer_ptr,%edx   # edx <- buffer_ptr
    lea (%edx,%ebx),%edx  # edx <- M[buffer_ptr] + offset
    mov %eax,(%edx,%ecx)  # dirección efectiva = M[buffer_ptr] + offset +
posi_color
    inc %ecx
    movb rojo,%al         # intensidad rojo = 2
    mov buffer_ptr,%edx   # edx <- buffer_ptr
    lea (%edx,%ebx),%edx  # edx <- M[buffer_ptr] + offset
    mov %eax,(%edx,%ecx)  # dirección efectiva = M[buffer_ptr] + offset +
posi_color
```

```
    ## siguiente fila
    jmp fila              # SALTO A fila -> bucle hasta llegar a salida

fil_exit:


################################################
### RESTAURO  EL CONTEXTO ANTERIOR A LA SUBRUTINA ###
    pop %esi
    pop %edi
    pop %edx
    pop %ecx
    pop %ebx
    pop %eax
################################################

    ## Recuperar el antiguo frame
    mov %ebp,%esp   # dejar donde estaba apuntando anteriormente esp
    pop %ebp        # registro ebp contiene direc a la que apuntaba inicialm.
    Ret             # Volver a rutina principal, programa .c

    .end            # fin de programa subrutina
```

Tras compilar el programa principal (incluyendo a la subrutina, cuya implementación se encuentra en pixels.s), se obtiene la siguiente imagen de tamaño 1024x1024, tal y como se ha definido inicialmente:

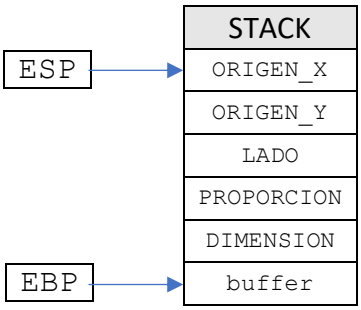
SALIDA BMP_AS.C



Lo que hemos hecho ha sido desde un programa codificado en el lenguaje de programación C, llamar a una subrutina (pixels_generator_2) la cual genera la matriz ya vista anteriormente. Tras obtener esta matriz, que contiene la información de la imagen deseada, se llamada a otra función para generar el fichero .bmp a partir de esta información.

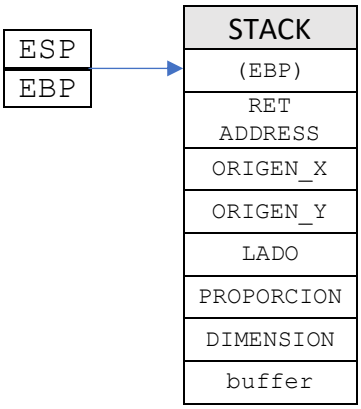
Cuando se hace la llamada a la subrutina pixels_generator_2, los argumentos se apilan en la Stack (pila) de la siguiente manera:

```
pixels_generator_2(ORIGEN_X,ORIGEN_Y,LADO,PROPORCION,DIMENSION,buffer);
```



Que, tras hacer la llamada a la subrutina, automáticamente se apila la dirección de retorno, quedando la pila de la siguiente manera:

Cabe destacar que, en el prólogo de la subrutina, además se apila la dirección a la que apuntaba EBP inicialmente para la creación de un nuevo frame. Este se apila para después poder capturarlo y dejar la pila como inicialmente se encontraba. Se muestra este cambio en la pila.



GDB

bmp_funcion.c

Se muestra la depuración correspondiente en el que se muestran los comandos utilizados para ver el contenido del frame pointer, así como el stack pointer y la dirección de retorno de la subrutina.

```
+file bmp_funcion          cargar programa bmp_funcion
Reading symbols from bmp_funcion...
+b main                    punto ruptura main
Punto de interrupción 1 at 0x1421: file bmp_funcion.c, line 126.
+layout Split              cargar instrucciones en ensamblador
+focus cmd                 enfocar a terminal
Focus set to cmd window.
+run                       ejecutar depuración
Starting program: /home/sayechu/Escritorio/bmp_funcion
[Depuración de hilo usando libthread_db enabled]
```

```
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0xffffd2a4) at bmp_funcion.c:126
+n     ejecutar siguiente instrucción (RGB_data buffer[top][top])
+n     ejecutar siguiente instrucción (memset(buffer, 0, sizeof(buffer)))
+si    step siguiente instrucción ensamblador (lea -0xc000c (%ebp),%eax)
+si    step siguiente instrucción ensamblador (push %eax)
+si    step siguiente instrucción ensamblador (push $0x200)
+si    step siguiente instrucción ensamblador (push $0x40)
+si    step siguiente instrucción ensamblador (push $0x40)
+x /4xw $esp volcar 4 palabras en formato hexadecimal a partir de ESP
0xfff3d1a0:  0x00000040   0x00000040   0x00000200   0xfff3d1cc
+si step siguiente instrucción ensamblador (call ©x56556310 <pixels
generator>)
pixels_generator (x=64, y=64, maximo=512, reg_mem=0xfff3d1cc) at
bmp_funcion.c:107
+x /1xw $esp volcar 1 palabra en formato hexadecimal a partir de ESP
0xfff3d19c:  0x5655645a   DIRECCION DE RETORNO de pixels_generator
+x /5xw $esp volcar 5 palabras en formato hexadecimal a partir de ESP
0xfff3d19c:  0x5655645a   0x00000040   0x00000040   0x00000200
0xfff3d1ac:  0xfff3d1cc
+si    step siguiente instrucción ensamblador (push %ebp)
+si    step siguiente instrucción ensamblador (mov %esp,%ebp)
+x /6xw $esp volcar 5 palabras en formato hexadecimal a partir de ESP
0xfff3d198:  0xffffd1d8   0x5655645a   0x00000040   0x00000040
0xfff3d1a8:  0x00000200   0xfff3d1cc
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

0xffffd1d8 ES LA DIRECCIÓN A LA QUE APUNTABA EBP INICIALMENTE, SE GUARDA PARA
CAPTURAR MÁS ADELANTE CUANDO SE VUELVA A RUTINA PRINCIPAL

0x5655645a ES LA DIRECCIÓN DE RETORNO DE LA SUBRUTINA pixels_generator

\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
+x /6xw $ebp EBP=ESP -> volcar contenido a partir EBP = volcar contenido a
partir ESP
0xfff3d198:  0xffffd1d8   0x5655645a   0x00000040   0x00000040
0xfff3d1a8:  0x00000200   0xfff3d1cc
+x /1xw ($esp+4)    volcar contenido en esp+4 -> dirección de retorno
0xfff3d19c:  0x5655645a
+x /1xw ($ebp+4)    volcar contenido en ebp+4 -> dirección de retorno
0xfff3d19c:  0x5655645a
+quit               SALIR DE DEPURACIÓN
```

Cuando se realiza la llamada a la función pixels_generator, antes de entrar en esa función y hacer las correspondientes instrucciones, se apilan los argumentos en la Stack, como se puede apreciar en la depuración.

Se muestran un par de capturas de la pantalla del GDB a continuación:

```
   0x56556442 <main+94>     add    $0x10,%esp
   0x56556445 <main+97>     lea    -0xc000c(%ebp),%eax
   0x5655644b <main+103>    push   %eax
   0x5655644c <main+104>    push   $0x200
   0x56556451 <main+109>    push   $0x40
   0x56556453 <main+111>    push   $0x40
 > 0x56556455 <main+113>    call   0x56556310 <pixels_generator>
   0x5655645a <main+118>    add    $0x10,%esp
   0x5655645d <main+121>    lea    -0xc000c(%ebp),%eax
   0x56556463 <main+127>    push   %eax
   0x56556464 <main+128>    push   $0x200
```

En esta primera captura se puede observar cómo se apilan los parámetros cuando se hace la llamada a la función.

```
 > 0x56556310 <pixels_generator>          push   %ebp
   0x56556311 <pixels_generator+1>        mov    %esp,%ebp
   0x56556313 <pixels_generator+3>        sub    $0x10,%esp
   0x56556316 <pixels_generator+6>        call   0x5655649d <__x86.get_pc_thunk.ax>
   0x5655631b <pixels_generator+11>       add    $0x2cad,%eax
   0x56556320 <pixels_generator+16>       mov    0x8(%ebp),%eax
   0x56556323 <pixels_generator+19>       mov    %eax,-0x8(%ebp)
   0x56556326 <pixels_generator+22>       jmp    0x565563c4 <pixels_generator+180>
   0x5655632b <pixels_generator+27>       mov    0xc(%ebp),%eax
   0x5655632e <pixels_generator+30>       mov    %eax,-0x4(%ebp)
   0x56556331 <pixels_generator+33>       jmp    0x565563a3 <pixels_generator+147>
   0x56556333 <pixels_generator+35>       mov    -0x8(%ebp),%edx
```

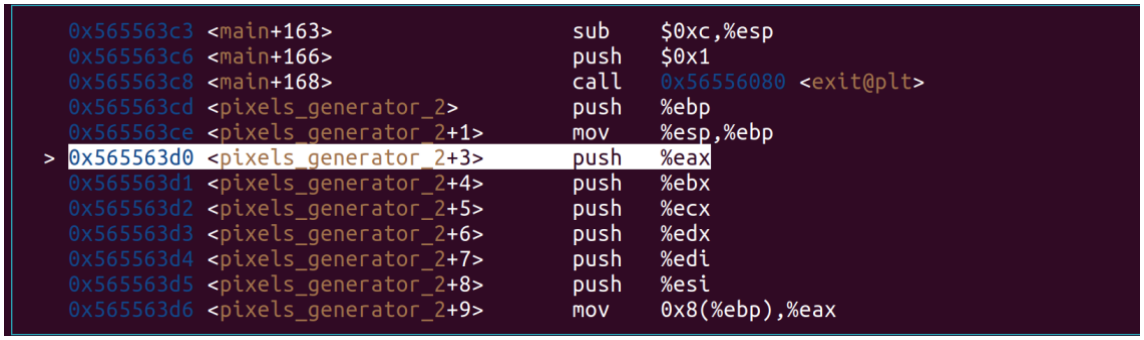
Esta captura muestra el código correspondiente a la función pixels_generator, ya después de haber hecho el call y haber apilado los parámetros en la pila.

bmp_as.c

```
+file bmp_as          cargar modulo Fuente a depurar, bmp_as
Reading symbols from bmp_as...
+b main               punto de interrupción en main
Punto de interrupción 1 at 0x135d: file bmp_as.c, line 117.
+layout Split         cargar pantalla con instrucciones en ensamblador
+focus cmd            enfocar a terminal
Focus set to cmd window.
+run                  ejecutar depuración programa
Starting program: /home/sayechu/Escritorio/bmp_as
[Depuración de hilo usando libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0xffffd2b4) at bmp_as.c:117
+n     ejecutar siguiente instrucción (RGB_data buffer[top][top])
+n     ejecutar siguiente instrucción (memset(buffer, 0, sizeof(buffer)))
+si    step siguiente instrucción ensamblador (sub $0x8,%esp)
+si    step siguiente instrucción ensamblador (lea -0x30000c (%ebp),%eax)
+si    step siguiente instrucción ensamblador (push %eax)
+si    step siguiente instrucción ensamblador (push $0x400)
+si    step siguiente instrucción ensamblador (push $0x7f)
+si    step siguiente instrucción ensamblador (push $0x200)
```

```
+si    step siguiente instrucción ensamblador (push 0x0)
+si    step siguiente instrucción ensamblador (push 0x0)
+x /6xw $esp volcar 6 palabras en formato hexadecimal a partir de ESP
0xffcfd1a0:  0x00000000   0x00000000   0x00000200   0x0000007f
0xffcfd1b0:  0x00000400   0xffcfd1dc
+si    step siguiente instrucción ensamblador (call 0x565563cd
<pixels generator 2>)
pixels_generator_2 () at pixels.s:61
+x /1xw $esp volcar 1 palabra en hexadecimal a partir de ESP -> dirección de
retorno de subrutina pixels_generator_2 a programa principal bmp_as.c
0xffcfd19c:  0x565563a0
+x /7xw $esp volcar 7 palabras en hex. a partir de ESP -> contenido pila
0xffcfd19c:  0x565563a0   0x00000000   0x00000000   0x00000200
0xffcfd1ac:  0x0000007f   0x00000400   0xffcfd1dc
+si    step siguiente instrucción ensamblador (push %ebp)
+si    step siguiente instrucción ensamblador (mov %esp,%ebp)
+x /8xw $esp volcar 8 palabras en hexadecimal a partir de ESP
0xffcfd198:  0xffffd1e8   0x565563a0   0x00000000   0x00000000
0xffcfd1a8:  0x00000200   0x0000007f   0x00000400   0xffcfd1dc
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

0xffffd1e8 ES LA DIRECCIÓN A LA QUE APUNTABA EBP INICIALMENTE, SE GUARDA PARA
CAPTURAR MÁS ADELANTE CUANDO SE VUELVA A RUTINA PRINCIPAL

0x565563a0 ES LA DIRECCIÓN DE RETORNO DE LA SUBRUTINA pixels_generator_2

\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
+x /8xw $ebp volcar 8 palabras en hex a partir de EBP;
0xffcfd198:  0xffffd1e8   0x565563a0   0x00000000   0x00000000
0xffcfd1a8:  0x00000200   0x0000007f   0x00000400   0xffcfd1dc
EBP y ESP apuntan a misma "celda" de la Stack, por lo tanto volcar el
contenido a partir de ESP o de EBP es indiferente.
+x /1xw ($esp+4) volcar 1w en hex a partir de ESP+4 -> dirección de retorno
0xffcfd19c:  0x565563a0
+x /1xw ($ebp+4) volcar 1w en hex a partir de EBP+4 -> dirección de retorno
0xffcfd19c:  0x565563a0
+quit         salir de depuración
```

Se muestran capturas de la pantalla de depuración de este último programa. Son de la parte del layout Split en la que se observan las instrucciones en ensamblador, que son las seguidas en la depuración del programa.



Comandos de Compilación

A continuación, se muestran los comandos usados para la compilación de los programas sobre los que trata esta práctica.

Cabe destacar que se ha generado un Makefile a partir del cual se van a hacer las compilaciones de todos los programas. Para ello, lo que vamos a hacer es copiar el programa que queremos compilar con el siguiente comando:

```
cp bmp_pixeles.c bmp_imagen.c
```

Con este comando lo que hacemos es generar otro programa igual pero con el nombre de bmp_imagen.c. Esto es para que nuestro Makefile nos reconozca el programa por el nombre de bmp_imagen.c

Por ejemplo, para el programa de bitmap_gen_test.c, lo haríamos de la siguiente manera:

```
cp bitmap_gen_test.c bmp_imagen.c
```

De esta forma, ya temenos el programa que queremos compilar con su respectivo nombre para que el Makefile sea capaz de reconocerlo. Ahora se procede a usar el makefile con el comando “make” en el respectivo directorio en el que se encuentra el programa. Esto nos genera el ejecutable listo para ser cargado en memoria.

Para el programa cuya implementación se encuentra codificada en ensamblador hay que compilar de una manera diferente. Para ello, procedemos a generar el módulo objeto a partir de la implementación de la función para poder compilar el programa principal incluyendo dicho objeto.

Para ello usamos el siguiente comando:

```
as --32 -gstabs -o pixels.o pixels.s
```

Esto nos genera el pixels.o. Se puede comprobar si se ha generado dicho fichero con el comando ls.
Una vez generado el módulo objeto, se procede a compilar el programa principal incluyendo dicho modulo. Se hace con el siguiente comando:

```
gcc -m32 -g -o bmp_as bmp_as.c pixels.o
```

Con este último comando se genera el ejecutable (`bmp_as`) con el que podremos generar la imagen. Para visualizar dicha imagen usamos el comando `display` junto con el nombre del fichero de dicha imagen. En este caso usaremos `display test_as.bmp`.

Conclusiones

Se comienza viendo y explicando el formato de las imágenes BitMap. Más adelante se ven diferentes programas, los cuales generan una imagen distinta. Se detalla el funcionamiento de los bucles que "dibujan" la imagen inicialmente en la matriz.

Posteriormente, dado un programa con una función, que es la que "dibuja", por decirlo de alguna manera, en la matriz los colores de la imagen que se quiere generar, se procede a codificar esa función en lenguaje ensamblador AT&T x86-32.

A lo largo de esta memoria se ha ido mostrando el contenido de la pila, tanto en la depuración como por dibujos en esta misma memoria, los cuales ayudan a entender el funcionamiento del programa tratado.