**PRÁCTICA 6: IMAGEN BIT MAP PORTABLE:**

**ÍNDICE**

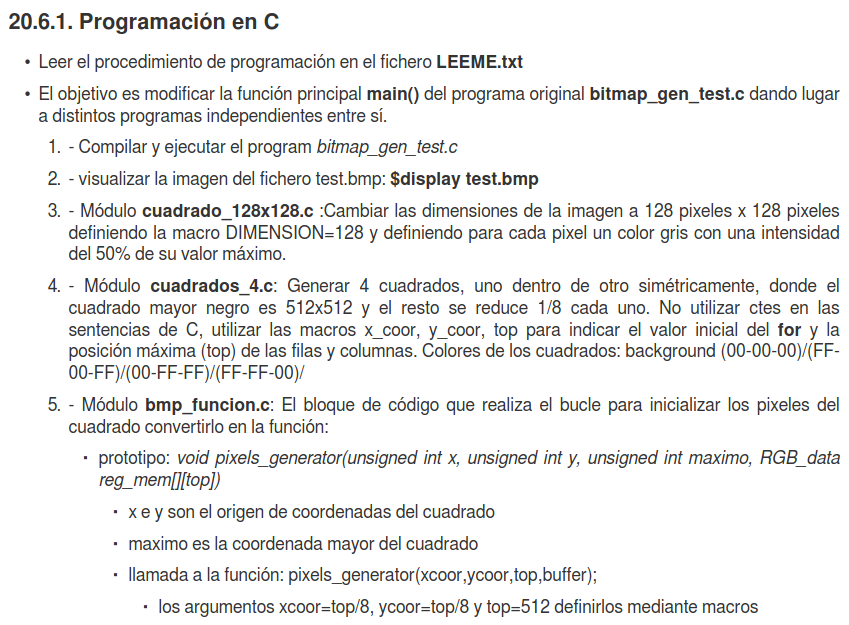
**Programación en C........................................................................................ 3**

**Programación en ASM ................................................................................................ 8**

**GDB ................................................................................................ 11**

**Comandos de Compilación ............................................................................................. 17**

**Conclusión ...........................................................................................................17**



Tras leer el procedimiento del fichero, se realizará los cambios en los distintos programas independientes.

En primer lugar, compilamos el programa bit\_map\_gen\_test.c. Para ello, utilizaremos el comando:

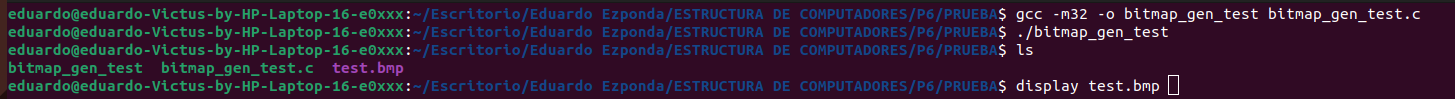
- gcc -m32 -o bitmap\_gen\_test bitmap\_gen\_test.c

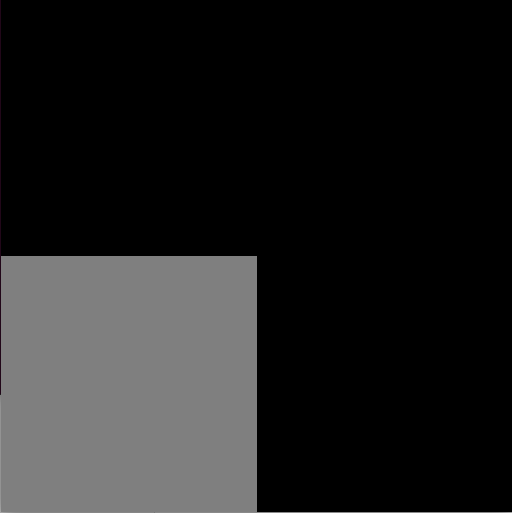
Para ejecutar bitmap\_gen\_test:

- ./bitmap\_gen\_test

Por último, para mostrar por pantalla el mapa de bits:

- display test.bmp





El mapa de bits de dimensiones 512x512 píxeles es de un

color grisáceo al utilizar una proporción 0x7f en cada uno

de los bytes del rgb\_data (blue, red, green). Si todos los

bytes tuvieran el valor de 0, el resultado sería el color negro

y por eso se muestra en el resto del mapa de bits.

En primer lugar, a través de la acción memset inicializamos

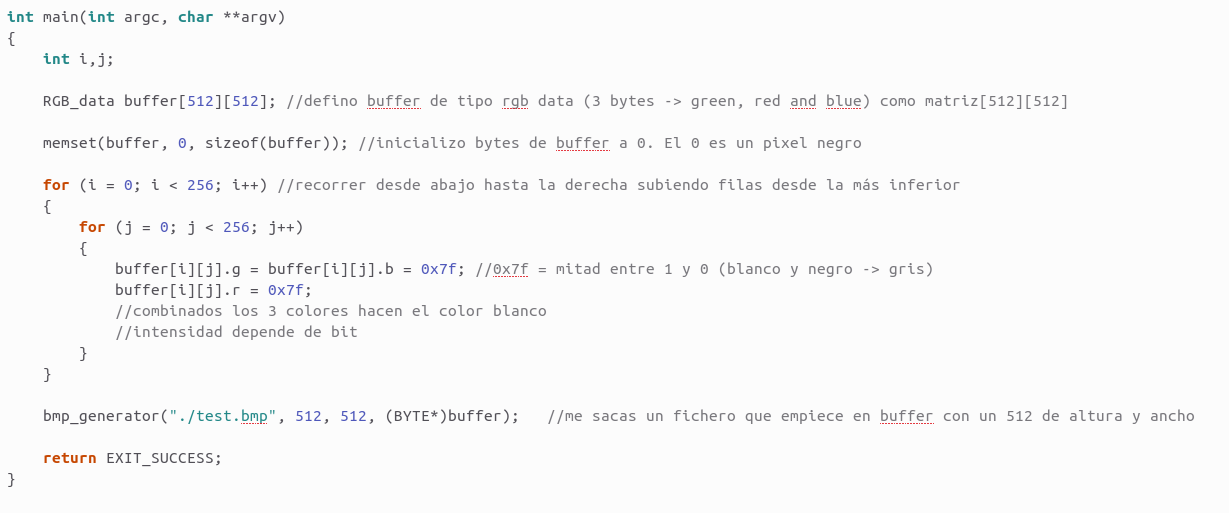
cada uno de los píxeles a 0 (negro).

A continuación, realizamos un doble bucle en el que la

primera mitad inferior izquierda se modifica el rgb\_data

para que cada byte tome el valor de 0x7f (gris).

Por último, generaremos el mapa de bits con la acción bmp\_generator.





**3. - Módulo cuadrado\_128x128.c :** Cambiar las dimensiones de la imagen a 128 pixeles x 128 pixeles definiendo la macro DIMENSION=128 y definiendo para cada pixel un color gris con una intensidad del 50% de su valor máximo.



Para definir la macro DIMENSIÓN, utilizamos la instrucción:

- #define DIMENSION 128



Por último, utilizamos la proporción 0x7f para una intensidad del 50% de su máximo para generar un tono grisáceo.

Por los datos del doble bucle, únicamente se modificarán la mitad inferior izquierda. El resto de los píxeles serán negros.

**4. - Módulo cuadrados\_4.c:** Generar 4 cuadrados, uno dentro de otro simétricamente, donde el cuadrado mayor negro es 512x512 y el resto se reduce 1/8 cada uno. No utilizar ctes en las  
sentencias de C, utilizar las macros x\_coor, y\_coor, top para indicar el valor inicial del for y la  
posición máxima (top) de las filas y columnas. Colores de los cuadrados: background (00-00-00)/(FF-00-FF)/(00-FF-FF)/(FF-FF-00)/



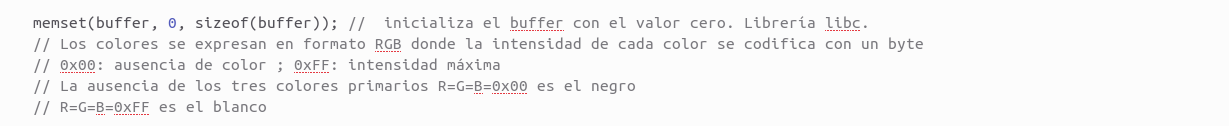
En primer lugar, definimos las macros a través de la instrucción #define.

A continuación realizaremos los cuatro cuadrados con su respectiva intensidad. El primer cuadrado negro (todos los valores a 0) se realiza con la acción memset.

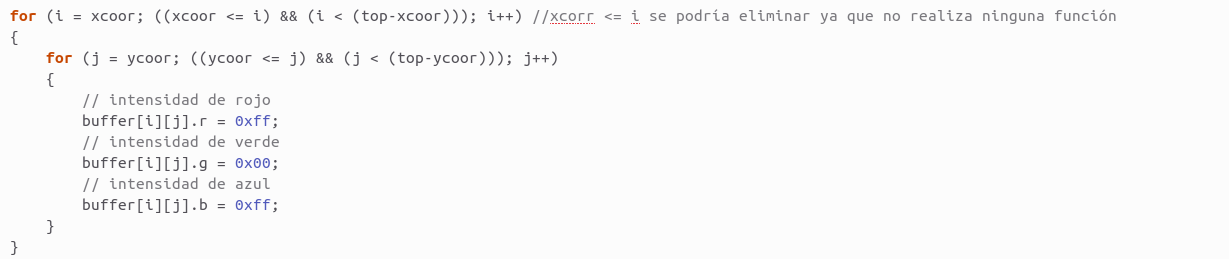
Luego, se realizarán los tres dobles bucles para cada uno de los colores, y en función de que tamaño del cuadrado le corresponda, el doble bucle tomará unas posiciones o otras.

Además, cada vez los tamaños de los cuadrados son más pequeños, con lo cual, el doble bucle tomará una menor cantidad de posiciones.

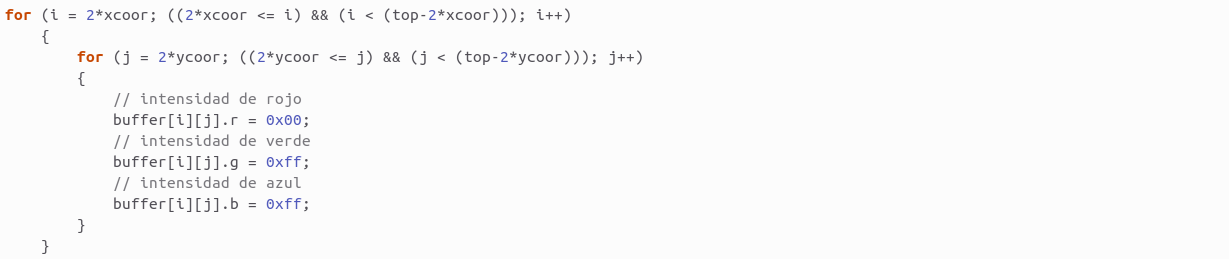
PRIMER CUADRADO: (rellena los 512 píxeles (0-511), pero más tarde se modifican)



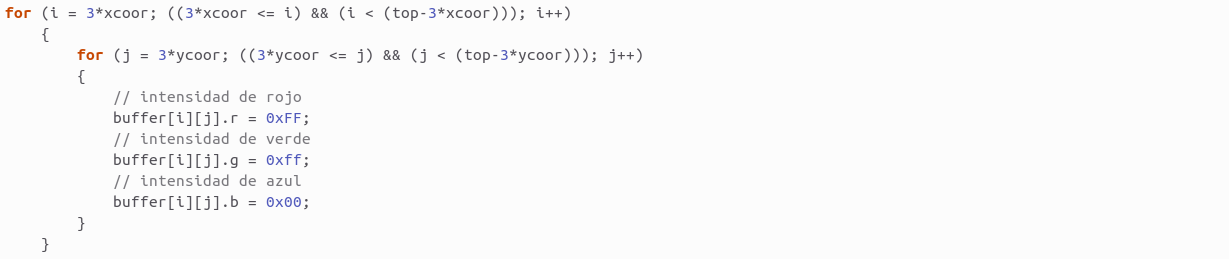
SEGUNDO CUADRADO: (rellena 64-447 píxeles)



TERCER CUADRADO: (rellena 128-383 píxeles)



CUARTO CUADRADO: (rellena 192-319 píxeles)

**5. - Módulo bmp\_funcion.c:** El bloque de código que realiza el bucle para inicializar los pixeles del cuadrado convertirlo en la función:

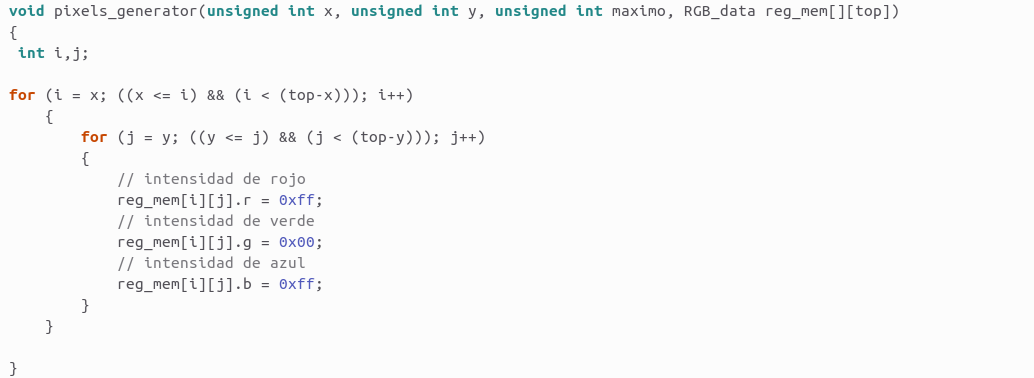
▪ prototipo: void pixels\_generator(unsigned int x, unsigned int y, unsigned int maximo, RGB\_data reg\_mem[][top])

▪ x e y son el origen de coordenadas del cuadrado

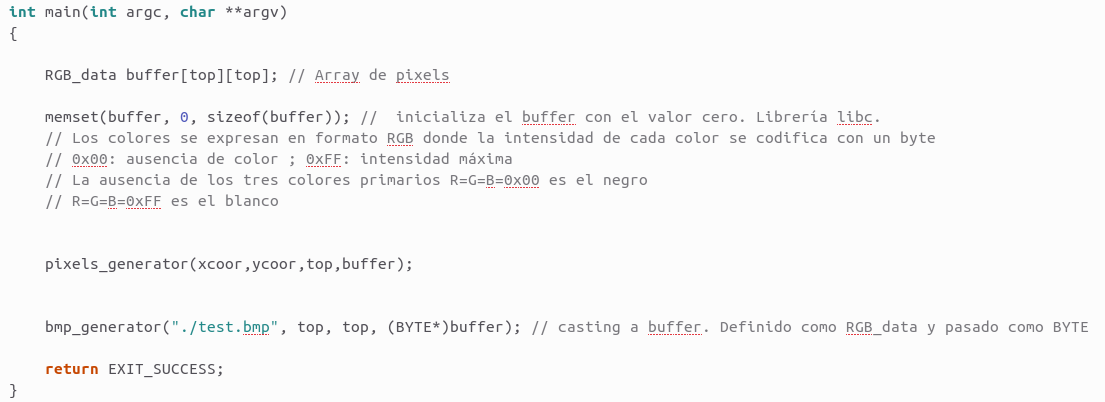
▪ maximo es la coordenada mayor del cuadrado

▪ llamada a la función: pixels\_generator(xcoor,ycoor,top,buffer);

▪ los argumentos xcoor=top/8, ycoor=top/8 y top=512 definirlos mediante macros

**ACCIÓN:**

**LLAMADA FUNCIÓN PASANDO LOS PARÁMETROS:**

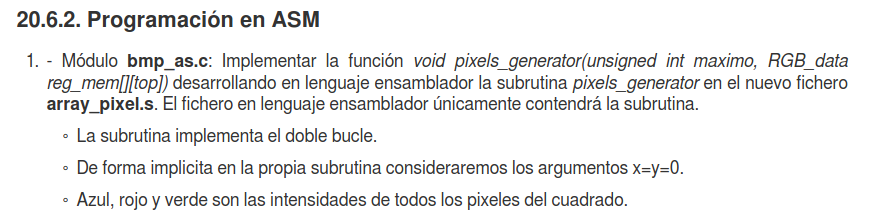


**DEFINICIÓN DE MACROS:**



Apunte:

Se podría modificar la inicialización de los píxeles a negro en vez de con la función memset con la función pixels\_generator. Para ello, únicamente se tendría que pasar como argumento la intensidad de los bytes, en este caso 0.



**LLAMADA FUNCIÓN:**

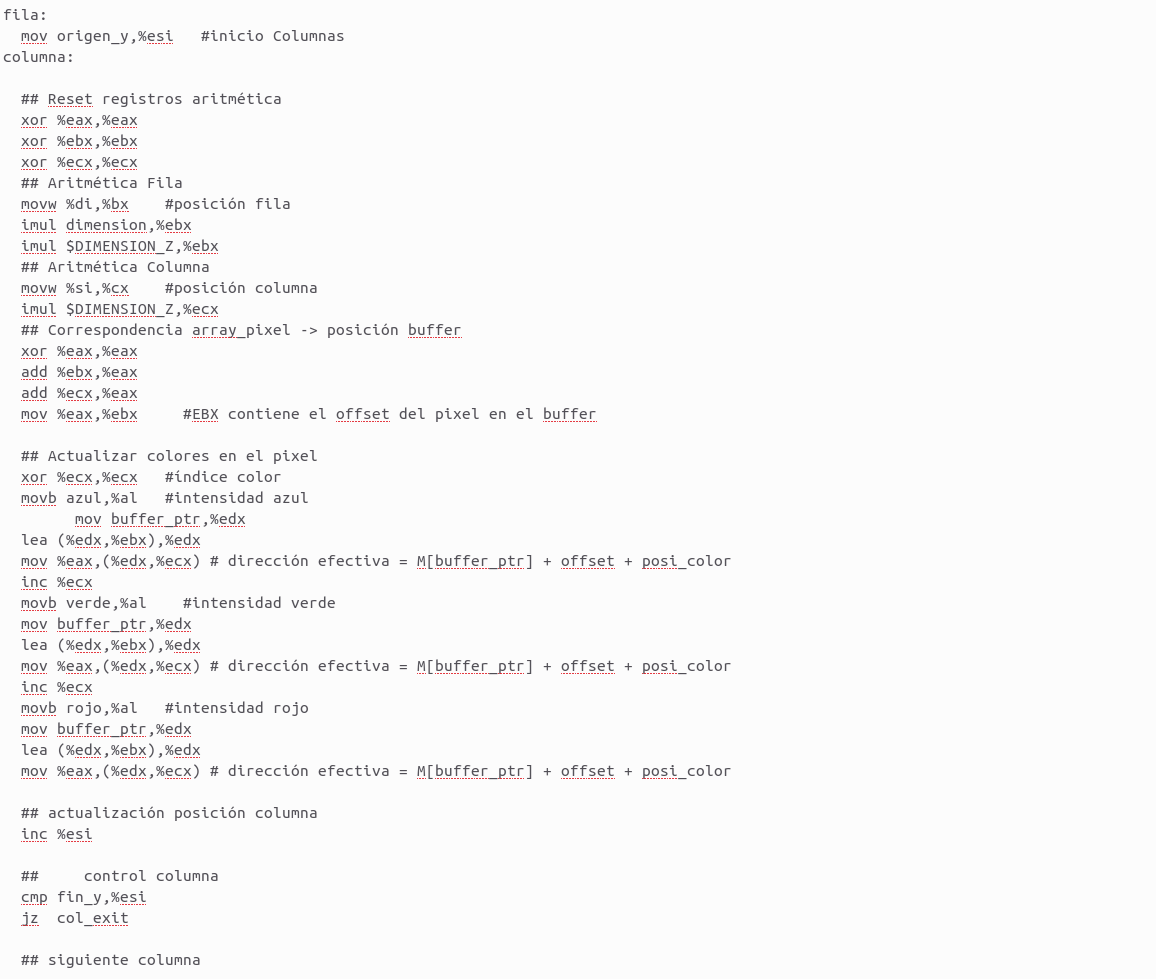


**DEFINICIÓN MACROS:**

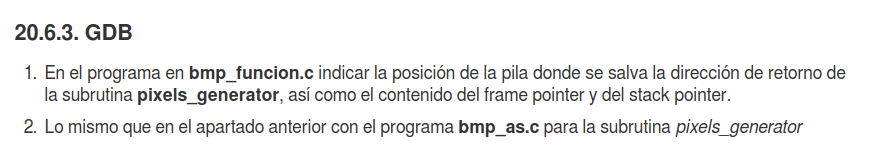


**ARCHIVO DE OBJETO ARRAY\_PIXEL.S:**





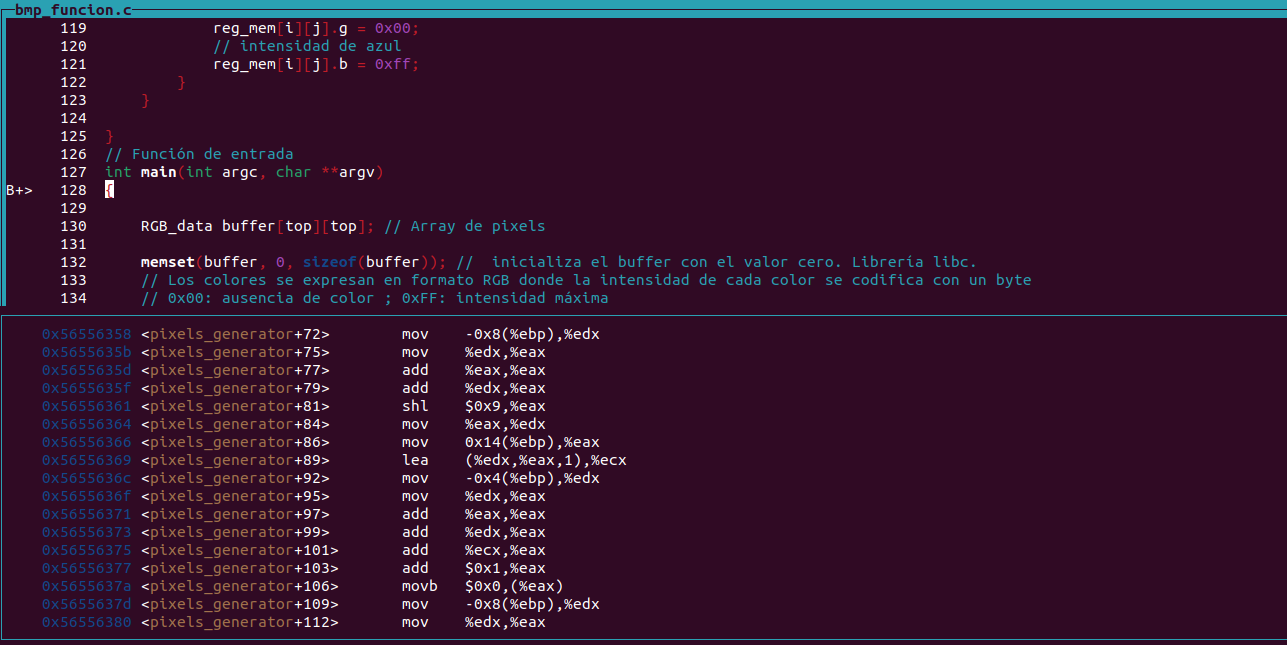




1. En primer lugar, abrimos el terminal y compilamos el programa bmp\_funcion.c con gcc -m32 -g -o bmp\_funcion bmp\_funcion.c . Después, nos introducimos en el gdb. Como estamos en un programa en lenguaje c y queremos ver la dirección de retorno y el contenido de la pila y punteros, utilizamos el comando layout split para ver las instrucciones en ensamblador.

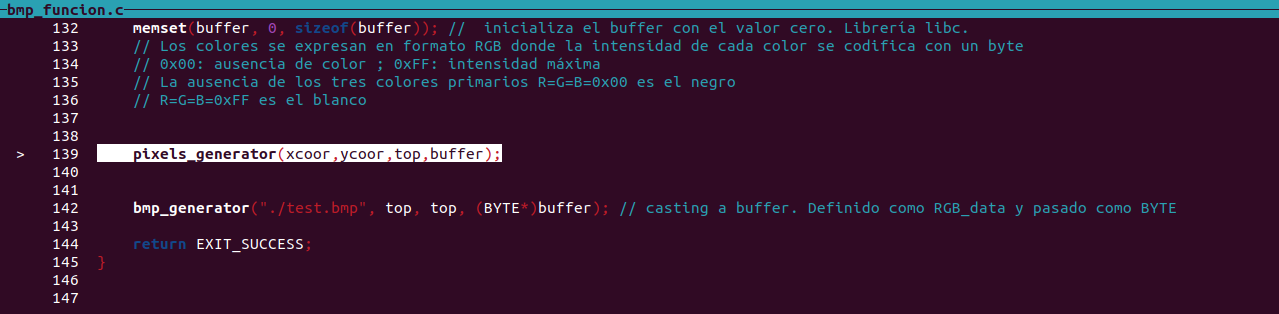
Para ello, deberíamos de haber introducido antes el breakpoint en main y haber hecho “run”.

Como se puede observar en la imagen, la mitad inferior sería el resultado de utilizar el comando layout split para mostrar todas las instrucciones en un nivel menor.

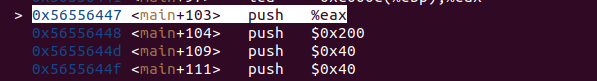


Utilizando el comando n (next) avanzaremos una instrucción en el programa.c, así como con el comando s (step) nos adentraremos en la función en el programa.c . Si queremos avanzar una a una las instrucciones en ensamblador, tendremos que utilizar ni (next instruction) y si (step instruction) para introducirnos en la función. Al realizar un next, se realizarán muchas instrucciones en ensamblador a la vez, a diferencia de que al realizar un next instruction, lo más probable es que se necesiten varios para pasar a la siguiente instrucción del programa.c .

Con lo cual, realizamos varios “next” hasta la instrucción de la llamada a la función pixels\_generator. Como se puede observar en la imagen, se realizan cuatro push correspondientes a los argumentos de la función en el orden inverso al ser la pila una estructura LIFO (Last In First Out). Realizando varios examine antes de la llamada a la función con call, podemos observar a través del stack pointer (esp) el contenido de los argumentos en la pila. El último, es la dirección del buffer.



Como se puede observar, nos encontramos en la instrucción de la llamada a la función pixels\_generator. Por ello, en la parte inferior, se apilan los 4 argumentos de la función.



**ARGUMENTOS FUNCIÓN:**

PUSH %EAX: eax contiene la dirección del buffer a modificar

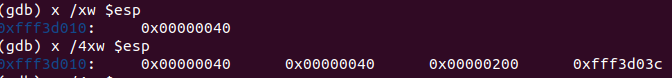
PUSH $0x200: contiene el valor de top en hexadecimal → 0x200 = 2\*16\*16 = 512 en decimal

PUSH $0x40: contiene el valor de ycoor en hexadecimal → 0x40 = 4\*16 = 64 en decimal

PUSH $0x40: contiene el valor de xcoor en hexadecimal → 0x40 = 4\*16 = 64 en decimal

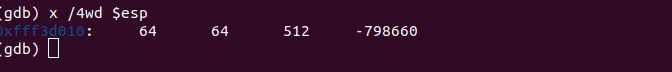


Realizando varios next instruction avanzamos hasta la llamada a pixels\_generator con call, para mostrar los argumentos de la pila con el comando examine.



En primer lugar, se muestra la cima de la pila en hexadecimal y tamaño word.

En la parte inferior, se muestran los cuatro argumentos de la función ya introducidos en la pila como se observa con el stack pointer. A continuación se muestran en decimal (d).

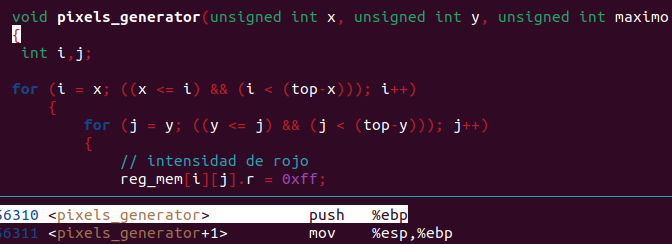


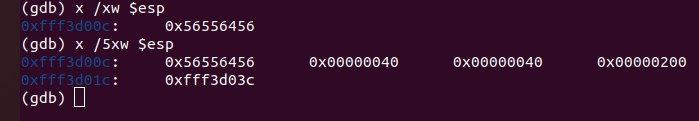
A continuación, introducimos el comando step instruction para llamar a la función.

A través de hacer print (p) $esp, mostramos la dirección de memoria donde está la cima de la pila, y por tanto de la dirección de retorno.



Como se observa, se introduce en la pila la dirección de retorno que en este caso es 0x56556456, que es el contenido del stack pointer. Los siguientes valores de la pila son los argumentos de la función.





Por último, se apila la antigua dirección del frame pointer (ebp) para tenerla guardada, y se crea el nuevo frame.

Los primer examine se realiza antes de hacer el mov %esp, %ebp en el que el frame pointer empieza a apuntar al stack pointer.



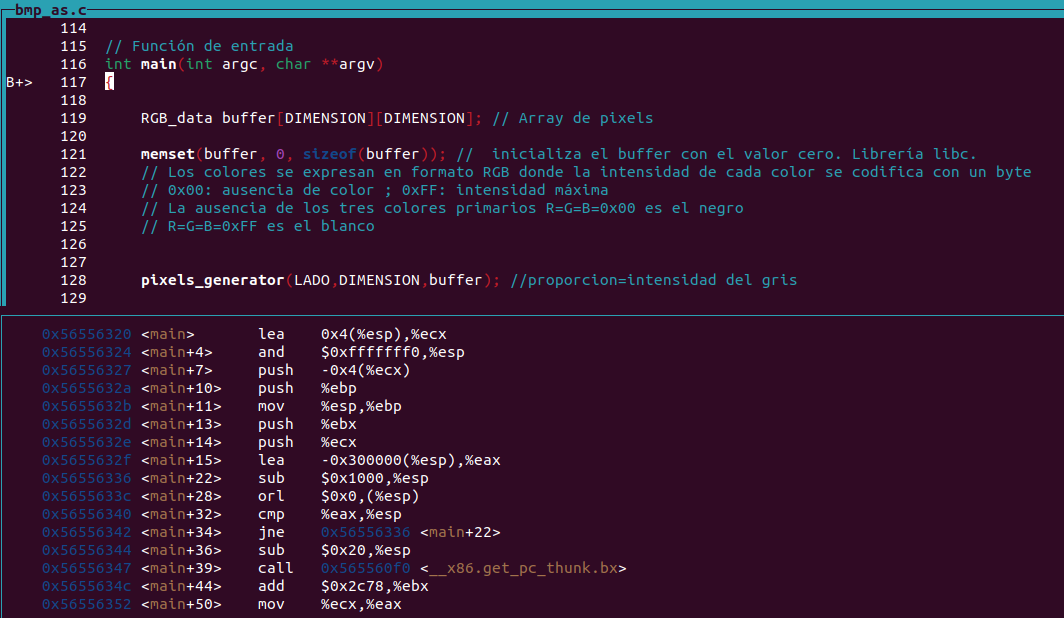
Realizamos el mismo procedimiento para el programa bmp\_as.c y la subrutina pixels\_generator.

1. En primer lugar, abrimos el terminal y compilamos el programa bmp\_as.c y el archivo de objeto as - - 32 – gstabs -o pixels\_generator.o pixels\_generator para después ejecutar compilar con gcc -m32 -g -o bmp\_funcion pixels\_generator.o bmp\_funcion.c .

Después, nos introducimos en el gdb. Como estamos en un programa en lenguaje c y queremos ver la dirección de retorno y el contenido de la pila y punteros, utilizamos el comando layout split para ver las instrucciones en ensamblador.

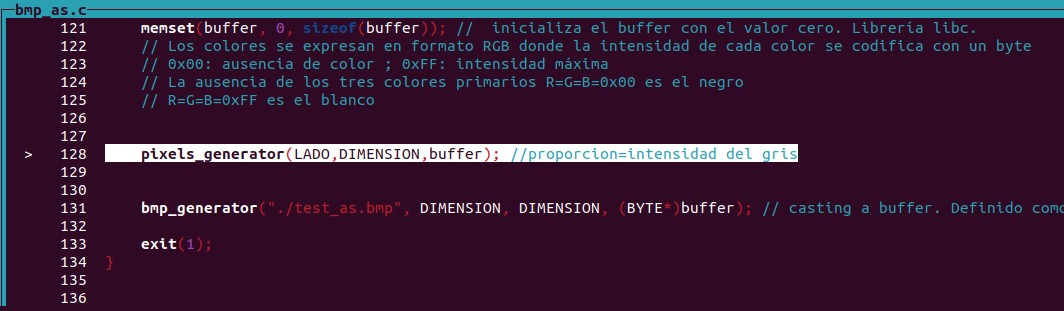
Para ello, deberíamos de haber introducido antes el breakpoint en main y haber hecho “run”.

Como se puede observar en la imagen, la mitad inferior sería el resultado de utilizar el comando layout split para mostrar todas las instrucciones en un nivel menor.

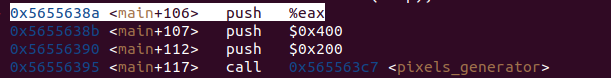


Utilizando el comando n (next) avanzaremos una instrucción en el programa.c, así como con el comando s (step) nos adentraremos en la función en el programa.c . Si queremos avanzar una a una las instrucciones en ensamblador, tendremos que utilizar ni (next instruction) y si (step instruction) para introducirnos en la función. Al realizar un next, se realizarán muchas instrucciones en ensamblador a la vez, a diferencia de que al realizar un next instruction, lo más probable es que se necesiten varios para pasar a la siguiente instructión del programa.c .

Con lo cual, realizamos varios “next” hasta la instrucción de la llamada a la función pixels\_generator. Como se puede observar en la imagen, se realizan tres push correspondientes a los argumentos de la función en el orden inverso al ser la pila una estructura LIFO (Last In First Out). Realizando varios examine antes de la llamada a la función con call, podemos observar a través del stack pointer (esp) el contenido de los argumentos en la pila. El último, es la dirección del buffer.



Como se puede observar, nos encontramos en la instrucción de la llamada a la función pixels\_generator. Por ello, en la parte inferior, se apilan los 3 argumentos de la función.



**ARGUMENTOS FUNCIÓN:**

PUSH %EAX: eax contiene la dirección del buffer a modificar

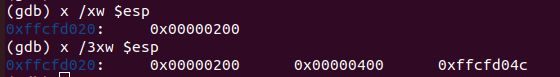
PUSH $0x400: contiene el valor de DIMENSION en hexadecimal → 0x400 = 4\*16\*16 = 1024 en decimal

PUSH $0x200: contiene el valor de LADO en hexadecimal → 0x200 = 2\*16\*16 = 512 en decimal

call $0x565563c7: llamada a la función pixels\_generator con la dirección de retorno

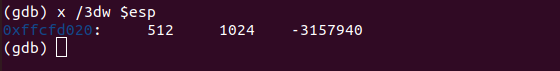


Realizando varios next instruction avanzamos hasta la llamada a pixels\_generator con call, para mostrar los argumentos de la pila con el comando examine.



En primer lugar, se muestra la cima de la pila en hexadecimal y tamaño word.

En la parte inferior, se muestran los tres argumentos de la función ya introducidos en la pila como se observa con el stack pointer. A continuación se muestran en decimal (d).

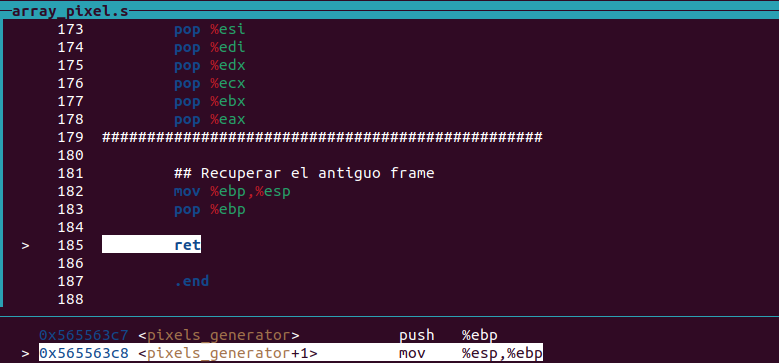


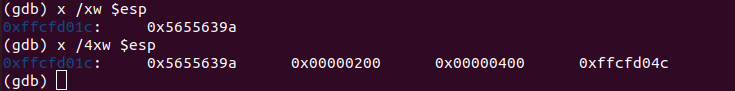
A continuación, introducimos el comando step instruction para llamar a la función.

A través de hacer print (p) /x (hexadecimal) $esp, mostramos la dirección de memoria donde se encuentra la cima de la pila y por tanto de la dirección de retorno.



Como se observa, se introduce en la pila la dirección de retorno que en este caso es 0x5655639a, que es el contenido del stack pointer. Los siguientes valores de la pila son los argumentos de la función.





Por último, se apila la antigua dirección del frame pointer (ebp) para tenerla guardada, y se crea el nuevo frame.

Los primer examine se realiza antes de hacer el mov %esp, %ebp en el que el frame pointer empieza a apuntar al stack pointer.



**COMPILACIÓN:**

gcc -m32 -g -o bitmap\_gen\_test bitmap\_gen\_test.c

gcc -m32 -g -o bmp\_as bmp\_as.c

gcc -m32 -g -o bmp\_funcion bmp\_funcion.c

gcc -m32 -g -o cuadrado\_128x128 cuadrado\_128x128.c

gcc -m32 -g -o cuadrados\_4 cuadrados\_4.c

as – - 32 -gstabs -o array\_pixel.o array\_pixel.s

Cabe destacar el uso de -m32 para usar una máquina de 32 bits, y -g para cargar la tabla de

símbolos. Además el comando “as – - 32 -gstabs -o array\_pixel.o array\_pixel.s” crea el archivo de objeto necesario para la ejecución de la función pixel\_generator en asm del programa bmp\_as.c

**CONCLUSIÓN:**

Durante ésta práctica se ha introducido la programación de los mapas de bits tanto en lenguaje c como en ASM. Para ello, es necesario la estructura RGB\_data que la conforman 3 bits distintos (red, blue, green), y en función de su valor, el píxel tomará un valor o otro.

Con la función memset, se inicializan todos los píxeles del mapa con un valor determinado.

Si los 3 valores del RGB\_data son 0x00, entonces dará lugar al negro. Sin embargo, si los valores son 0xFF, el resultado será blanco.

Por último, un valor medio como 0x7f dará un tono grisáceo, al tener un valor entre el negro 0x00 y el blanco 0xFF.

Además, con la función bmp\_generator se generará el mapa de bits, que una vez compilado y ejecutado el programa, con el comando display se mostrará por pantalla.

Para la programación en ASM de la función pixels\_generator, es necesario pasar los argumentos a través de la pila para después capturarlos.

Además, el contenido de la pila tiene que ser igual al final de la función que antes de la llamada, y por ello se guardan los registros y la dirección del bottom pointer en la pila para crear a continuación el nuevo frame.

Para crear el doble bucle, es necesario el uso de dos etiquetas y dos registros que irán aumentando su valor hasta que adquieran el valor del final de la fila o columna.

En el caso de adquiera el valor de la última fila y columna, el recorrido habrá finalizado.

Para calcular el offset de la fila a modificar, es necesario realizar las siguientes instrucciones:

movw %di,%bx #posición fila

imul dimension,%ebx

imul $DIMENSION\_Z,%ebx

El registro %edi contiene la posición del a fila a modificar. A partir de ello, se le realizará la multiplicación entera con dimensión (número de píxeles) y DIMENSIÓN\_Z (3, bytes structura RGB\_data (green, red, blue)).

Para calcular el offset de la columna modificar, es necesario realizar las siguientes instrucciones:

movw %si,%cx #posición columna

imul $DIMENSION\_Z,%ecx

En este caso, el registro %esi contiene la posición de la columna. Una vez obtenidos ambos valores, su suma dará lugar al offset.

Finalmente, para el cálculo de la dirección efectiva en cada byte:

movb verde,%al #intensidad verde

mov buffer\_ptr,%edx

lea (%edx,%ebx),%edx

mov %eax,(%edx,%ecx) # dirección efectiva = M[buffer\_ptr] + offset + posi\_color

inc %ecx

El registro ebx contiene el offset.

El registro edx contiene M[buffer\_ptr].

El registro %al contiene posi\_color.

Por último, es necesario incrementar ecx para que pase al siguiente byte del struct RGB\_data.

El orden de declaración de la estructura RGB\_data tiene que concordar con el valor de ecx en este caso.

Para finalizar el programa, se restablecen los valores de los registros y se recupera el antiguo frame debido a que la dirección de ebp estaba apilada anteriormente en la pila.