Administración y Organización de Computadores

Curso 2022-2023

<u>Práctica</u>: *Tetris*

El principal objetivo de esta práctica es completar una aplicación que implementa el juego *Tetris*. La parte de la aplicación que se encuentra ya implementada está escrita en *C++* y se encarga del control principal del juego y de la gestión de la interfaz de usuario. No obstante, este código se apoya en una serie de funciones que se encuentran vacías y cuya implementación es objeto de esta práctica. Dicha implementación debe realizarse en lenguaje ensamblador. La integración de los dos lenguajes se llevará a cabo mediante las facilidades de ensamblado en línea proporcionadas por el compilador *gcc*. La programación en ensamblador se realizará considerando la arquitectura de un procesador Intel o compatible de 64 bits.

A continuación, se detallan diferentes aspectos a tener en cuenta para el desarrollo de esta práctica.

<u>Descripción de la aplicación</u>

El código que deberá ser completado estará incluido en una aplicación *C++*, disponible como proyecto de *Qt*. La siguiente figura muestra una captura de pantalla en un instante de ejecución de la aplicación:

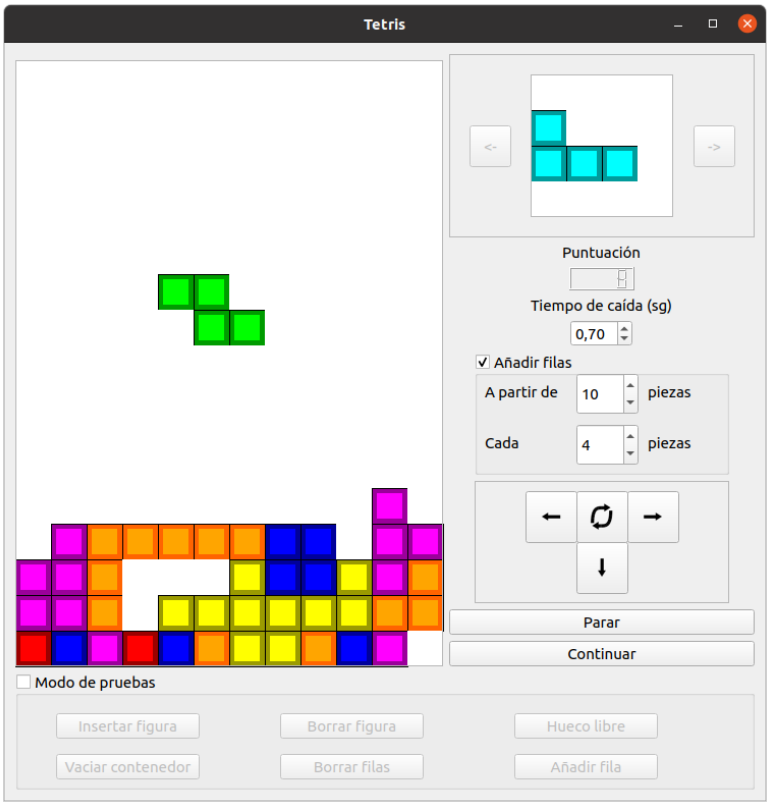

Figura 1: interfaz de la aplicación

¿Qué es Qt?

Qt es un *framework* de desarrollo de aplicaciones que, entre otras aportaciones, proporciona herramientas y librerías de clases para la creación de interfaces de usuario en entornos de escritorio.

Puesta en marcha del proyecto

El fichero que contiene la descripción del proyecto (*pracaoc.pro*) se encuentra disponible en la carpeta principal de la aplicación. Dicho fichero puede ser utilizado para importar el proyecto desde diferentes entornos de desarrollo. No obstante, se recomienda trabajar con Qt Creator (paquete **qtcreator**). Además del paquete **qtcreator**, es necesario instalar los paquetes que contienen las librerías y herramientas de Qt: **qttools5-dev-tools**, **qtbase5-dev**, **qtchooser, qt5-qmake** y **qtbase5-dev-tools**.

Funcionamiento de la aplicación

En el juego del *Tetris* intervienen una serie de piezas formadas por 4 bloques cuadrados. Estas piezas van cayendo durante el curso del juego en una matriz rectangular que denominaremos "contenedor". El usuario puede girar, mover a derecha o izquierda o forzar la caída de estas piezas hasta apoyarlas en la parte inferior del contenedor o sobre otros bloques ya situados. Cuando una fila del contenedor se completa, ésta se borra y las filas situadas sobre ella bajan una posición. El objetivo del juego es eliminar el mayor número posible de filas y evitar que el contenedor se complete y no puedan entrar nuevas piezas.

La interfaz principal de la aplicación se muestra en la figura 1. El área rectangular de la izquierda representa el contenedor sobre el que caen las piezas. En la parte superior derecha se muestra la siguiente pieza que caerá en el contenedor una vez que se sitúe la actual. Debajo de este elemento de la interfaz, se muestra la puntuación obtenida por el usuario hasta el momento. Por cada fila eliminada, el jugador consigue 1 punto. Los restantes elementos de la derecha de la interfaz están asociados con diferentes controles. El primero de ellos es el "Tiempo de caída" desde el cual el usuario puede seleccionar el tiempo que transcurre desde que la pieza se sitúa en la fila actual hasta que baja automáticamente a la siguiente fila. El siguiente grupo de controles se activa desde el cuadro de chequeo "Añadir filas". Este grupo de controles permite que el usuario añada una dificultad extra al juego insertando en la parte inferior del contenedor una nueva fila cada cierto número de piezas, una vez que se hayan colocado un determinado número de piezas. Debajo de este grupo de control, se encuentran los botones de movimiento de la pieza actual. A través de ellos el usuario puede girar la pieza o moverla una posición hacia la izquierda, hacia la derecha o hacia abajo. Los dos últimos botones permiten que el usuario comience un nuevo juego o finalice el actual y que pause o continúe el juego actual.

Además de estas opciones, en la parte inferior de la interfaz hay un grupo adicional de controles que permiten probar individualmente las distintas funciones que se ejecutan durante el transcurso del juego. Estas funciones son:

- *Insertar figura*: inserta una figura en la posición del contenedor indicada por el usuario. La figura se puede seleccionar a través del visor de siguiente pieza, utilizando los botones situados a los lados del visor. Es posible también rotar la figura mediante el botón para girar pieza utilizado durante el juego. El área donde se inserta la figura es la marcada por el cuadro negro que aparece en el contenedor (ver figura 2). Se puede cambiar dicho área, haciendo clic con el ratón sobre otra posición del contenedor.
- *Borrar figura*: borra la figura actualmente indicada en el visor de siguiente pieza dentro del área seleccionada del contenedor.
- *Hueco libre*: comprueba si existe hueco libre para colocar la figura actual del visor de siguiente pieza en el área seleccionada del contenedor. Si existe hueco libre, el cuadrado

que marca el área seleccionada del contenedor aparece en verde y, en caso contrario, en rojo.

- *Vaciar contenedor*: borra todo el contenido del contenedor.
- *Borrar filas*: borra las filas completas del contenedor.
- *Añadir filas*: añade una nueva fila de bloques en la parte inferior del contenedor.

Cada una de estas funciones se corresponden con los procedimientos que hay que implementar en la práctica para que el juego funcione correctamente. A través de estos botones es posible comprobar el correcto funcionamiento de los procedimientos de manera individual.

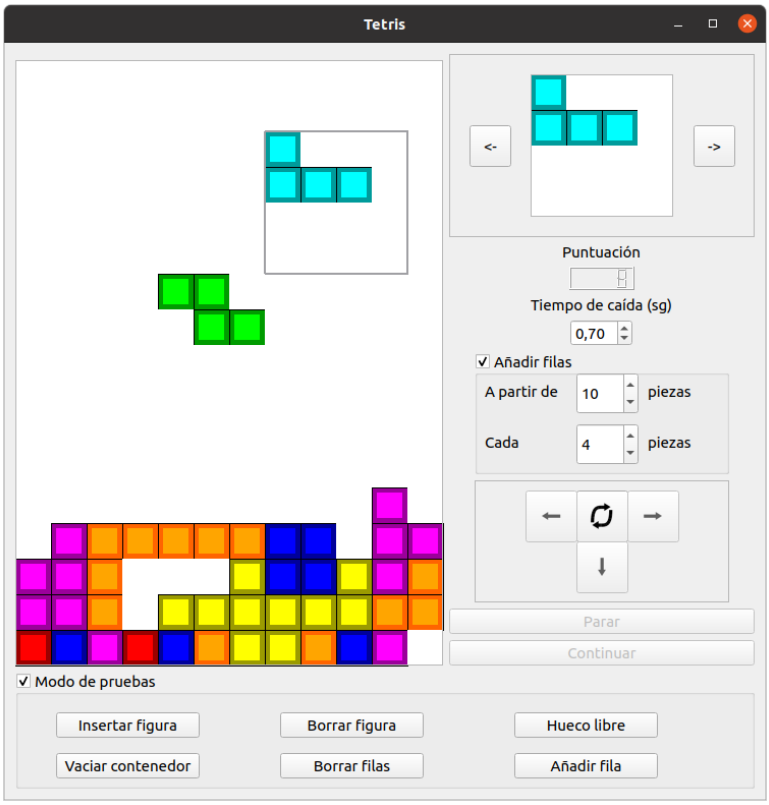

Figura 2: modo de pruebas (resultado de utilizar el botón “Insertar figura”)

Componentes de la aplicación

El código fuente de la aplicación está formado por 5 ficheros: *main.cpp*, *pracoc.cpp*, *pracoc.h*, *tetris.cpp* y *tetris.h*. Además, se incluye un formulario de Qt (*mainForm.ui*) y el fichero de descripción del proyecto (*pracaoc.pro*). El contenido de cada fichero fuente es el siguiente:

- *main.cpp*: contiene el procedimiento principal que permite lanzar la aplicación, así como crear y mostrar la ventana principal que actúa como interfaz entre el usuario y la aplicación.
- *pracaoc.h*: fichero que contiene la definición de la clase principal de la aplicación (*pracAOC*). Esta clase contiene los elementos principales de gestión de la aplicación. Entre los atributos, se encuentran la interfaz de usuario incluida en el programa y la variable que representa al contenedor del juego. Esta última (*container*) está definida como un array de tipo char de *CONTAINERH*CONTAINERW* elementos (el número de filas está definido en la constante *CONTAINERH* y el número de columnas por *CONTAINERW*). Cada elemento de este array almacena el contenido de la celda correspondiente del contenedor. **Los posibles valores de una celda son 0, indicando que dicha posición del contenedor se encuentra vacía, o un valor entre 1 y 7, que representa el color del bloque situado en esa celda**. La representación del contenedor del juego a través de este array supone un almacenamiento en memoria por filas. Esto implica que el acceso a una determinada celda situada en una fila *f* y una columna *c* del

contenedor se realiza a través de la posición del array *f*w+c*, siendo *w* el número de columnas del contenedor. La fila 0 representa la fila superior del contenedor y la columna 0, la columna de la izquierda. Para evitar tener que comprobar los límites del contenedor en la implementación de las distintas funciones, el array cuenta con 4 columnas y 4 filas más de las que son visibles en el juego. En concreto, hay dos columnas adicionales a cada lado del array y 4 filas en la parte inferior. El contenido de estas celdas es siempre 1, indicando de esta manera que esas posiciones del contenedor se encuentran ocupadas.

- *pracaoc.cpp*: Incluye la implementación de los métodos de la clase *pracAOC*. En su mayoría, estos métodos se encargan de responder a los distintos eventos de la interfaz de usuario incluida en el programa y de resolver la gestión del juego mediante llamadas a las funciones definidas en *tetris.h*. Más concretamente, la gestión del juego se resuelve de la siguiente manera:
  - Cuando comienza un nuevo juego, el contenedor debe vaciarse. Esto se lleva a cabo mediante una llamada a la función ***vaciarContenedor*** definida en *tetris.h*.
  - Durante el juego, cuando una nueva pieza debe introducirse en el contenedor, en primer lugar se comprueba si existe hueco libre para insertar la pieza en la posición de entrada del contenedor. Para ello, se realiza una llamada a la función ***huecoLibre*** definida en *tetris.h*, pasando, entre otros parámetros, la pieza actual, representada a través de una matriz de tamaño 4x4, y la posición (fila y columna) de entrada del contenedor. Al igual que en el contenedor, cada elemento de la matriz que representa la pieza tiene dos posibles valores: 0, para indicar que la celda no forma parte de la pieza, y un valor entre 1 y 7 que representa que la celda está presente como bloque de la pieza con un determinado color. El procedimiento ***huecoLibre***, a partir de la información recibida, comprueba si hay posibilidad de insertar la pieza en la posición especificada del contenedor y devuelve el valor *true* en ese caso. En caso contrario, el valor retornado por la función es *false*. En el caso de determinar la posibilidad de insertar la pieza, ésta se almacena en el contenedor, en la posición correspondiente, mediante una llamada al procedimiento ***insertaPieza*** definido en *tetris.h*. Si no hay hueco libre para introducir la pieza en el contenedor, el juego finaliza.
  - Si ya hay una pieza cayendo en el contenedor, por cada movimiento de la pieza (incluido el giro), se comprueba, inicialmente, si es posible situar la pieza en la nueva posición. Para ello, la pieza es primeramente borrada del contenedor, mediante una llamada al procedimiento ***borraPieza*** de *tetris.h*, y, a continuación, se comprueba la existencia de hueco libre en la nueva posición para esa pieza (llamando a la función ***huecoLibre*** de *tetris.h*). Si existe hueco libre, la pieza se inserta en la nueva posición del contenedor y, en caso contrario, en la posición actual (llamada al procedimiento ***insertaPieza***). Si no existiera hueco para insertar la pieza en la nueva posición, en el caso de que el movimiento de la pieza fuera hacia abajo, se asume que la pieza se ha situado en su posición final. En esta situación, se comprueba si existen filas completas que puedan ser eliminadas, mediante una llamada a la función ***borraFilas*** de *tetris.h*. Esta función devuelve el número de filas eliminadas, valor utilizado para incrementar la puntuación del jugador.
  - Una vez que la pieza actual está situada en el contenedor, siempre que la opción de "Añadir filas" esté activa, se realiza lo siguiente: si ya se han situado un número determinado de piezas (indicado en el elemento de la interfaz "*A partir de … piezas*") y han caído *n* piezas desde que se borro la última fila (siendo *n* el valor indicado en el elemento de la interfaz "*Cada … piezas*"), se inserta una nueva fila formada por bloque aleatorios en la parte inferior del contenedor, provocando que todo el contenido del contenedor suba una posición en fila. Esto se resuelve mediante una llamada a la función ***annadeFila***, definida en *tetris.h*.
- *tetris.h*: contiene la definición de las funciones implementadas en el fichero *tetris.cpp*.
- *tetris.cpp*: implementación de las funciones de control del juego invocadas a través de la clase principal de la aplicación (*pracAOC*). Todas estas funciones, exceptuando

vaciaContenedor, contienen una implementación vacía. El objetivo de esta práctica es completarlas para que el funcionamiento de la aplicación sea el descrito anteriormente.

Extensiones principales en x86-64

- En relación a la representación de datos en memoria, en 64 bits, los punteros y los datos de tipo *long* ocupan 64 bits. El resto de tipos mantiene el mismo tamaño que en la línea de procesadores de 32 bits (int: 4 bytes, short: 2 bytes, …).
- Todas las instrucciones de 32 bits pueden utilizarse ahora con operandos de 64 bits. En ensamblador, el sufijo de instrucción para operandos de 64 bits es "q".
- Los registros de 32 bits se extienden a 64. Para hacer referencia a ellos desde un programa en lenguaje ensamblador, hay que sustituir la letra inicial "e" por "r" (%rax, %rbx, …). Los registros de 32 bits de la IA32 se corresponden con los 32 bits de menor peso de estos nuevos registros.
- El byte bajo de los registros %rsi, %rdi, %rsp y %rbp es accesible. Desde el lenguaje ensamblador, el acceso a estos registros se realiza a través del nombre del registro de 16 bits finalizado con la letra "l" (%sil, %dil, …).
- Aparecen 8 nuevos registros de propósito general de 64 bits (%r8, %r9, …, %r15). Es posible acceder a los 4, 2 o al último byte de estos registros incluyendo en su nombre el sufijo d, w o b (%r8d – 4 bytes, %r8w – 2 bytes, %r8b – 1 byte).

Ejemplo de implementación en ensamblador x86-64 de una función de C/C++

Dentro del fichero *tetris.cpp*, se ha incluido la implementación de la primera función (*vaciaContenedor*) a modo de ejemplo. Como ya se ha comentado anteriormente, la implementación de las restantes es objeto de esta práctica. En algunos de los procedimientos, aparecen 2 bloques *asm* para poder incluir 2 implementaciones. La compilación de uno u otro bloque se realiza comentando o descomentando la sentencia "#define WITH_SSE" que hay al principio del fichero. En el primer bloque se incluirá la implementación obligatoria que utilizará instrucciones de CPU realizando un procesamiento secuencial. En el segundo bloque, se incluirá una implementación opcional utilizando un procesamiento vectorizado mediante instrucción SSE. Para compilar el código con esta segunda implementación deberá descomentarse la línea "#define WITH_SSE".

```
void tetris::vaciaContenedor(char * container, int w, int h)
{
#ifndef WITH_SSE
        asm volatile(

        "mov %0, %%rsi;"
        "mov %1, %%eax;"
        "mov %%eax, %%ebx;"
        "sub $2, %%ebx;"
        "mov $0, %%rcx;"
        "mov %2, %%ecx;"
        "BVaciarF:"
            "mov $0, %%edi;"
            "BVaciarC:"
                "cmp $2, %%edi;"
                "jl sigVaciar;"
                "cmp %%ebx, %%edi;"
                "jge sigVaciar;"
                "movb $0, (%%rsi);"
            "sigVaciar:"
                "inc %%rsi;"
                "inc %%edi;"
                "cmp %%eax, %%edi;"
                "jl BVaciarC;"
            "loop BVaciarF;"
        :
        : "m" (container), "m" (w), "m" (h)
        : "%rax", "%rbx", "%rcx", "%rsi", "%rdi", "memory"
```

```
    );
#else
    asm volatile(
        // Por simplificación, asumimos w=16.
        // Dicho valor es el utilizado como anchura
       // en la definición de "container"

        "mov %0, %%rsi;"
        "mov $0, %%rcx;"
        "mov %2, %%ecx;"

        "pxor %%xmm0, %%xmm0;"
        "mov $0xFFFF0000, %%eax;"
        "pinsrd $3, %%eax, %%xmm0;"
        "mov $0x0000FFFF, %%eax;"
        "pinsrd $0, %%eax, %%xmm0;"

        "BVaciarF:"
            "movdqu (%%rsi), %%xmm1;"
            "pand %%xmm0, %%xmm1;"
            "movdqu %%xmm1, (%%rsi);"

            "add $16, %%rsi;"
            "loop BVaciarF;"
        :
        : "m" (container), "m" (w), "m" (h)
        : "%rax","%rcx","%rsi","%xmm0","%xmm1","memory"
    );
#endif
}
```

La función *vaciaContenedor* incluye como parámetros un puntero al contenedor (debe tratarse como un array), el número completo de columnas que contiene (incluyendo las 4 columnas adicionales) y el número de filas visibles (sin incluir las 4 adicionales). Para tener acceso a estos parámetros desde el bloque de código en ensamblador, están incluidos como operandos de entrada (segunda lista situada al final del bloque, precedida de ":"). Tras la definición de estos operandos, se incluye la lista de registros utilizados dentro del código. Además de los registros indicados, dado que la memoria es modificada, la lista incluye también la palabra "memory".

Una vez aclaradas estas definiciones, analicemos a continuación paso a paso el código ensamblador incluido en el primer bloque *asm* del procedimiento:

- La primera instrucción se encarga de copiar la dirección inicial de memoria del array contenedor, indicada por el operando %0, en el registro %rsi para su posterior direccionamiento. Así, a través del registro %rsi podremos acceder a cada una de las celdas del contenedor. La dirección inicial se corresponde con la dirección de la primera celda.
- A continuación, se inicializa %eax con el número de columnas del contenedor, asignando el parámetro *w*, que se encuentra accesible a través del operando %1, a dicho registro.
- Para evitar borrar las celdas no visibles que actúan de "muros" del contenedor, el registro %ebx es inicializado a *w-2* mediante las dos siguientes instrucciones.
- Para controlar el recorrido por filas durante el proceso de vaciado del contenedor, se utiliza la instrucción *loop*, por lo que, lo siguiente que hace el procedimiento es inicializar %rcx con el número de filas visibles del contenedor (parámetro *h* accesible mediante el operando %2). Puesto que el parámetro *h* es de tipo *int*, esta inicialización se lleva a cabo en 2 partes. En primer lugar, se pone a 0 todo el registro %rcx y, a continuación, se realiza la asignación de *h* al registro %ecx (32 bits de menor peso de %rcx). Una vez hecha esta inicialización, comienza el bucle que recorre el contenedor por filas. Dicho bucle está delimitado por la etiqueta "*BVaciarF*" y la instrucción *loop* a dicha etiqueta.
- Dentro del recorrido por filas, se realiza un recorrido por columnas. Para ello, el registro %edi se utiliza como índice de columna, por lo que es inicializado a 0. Tras esta

inicialización, se sitúa el bucle que recorre el contenedor por columnas. Este bucle está delimitado por la etiqueta "*BVaciarC*" y el salto "*jl*" a dicha etiqueta.

- En el recorrido por columnas, en primer lugar se comprueba si la columna de la celda actual está dentro de las celdas visibles. Así, en el caso de que la celda esté situada en las 2 primeras o dos últimas columnas del contenedor, no hay que realizar el borrado y se pasa directamente a la siguiente iteración (marcada por la etiqueta "*sigVaciar*"). Si la celda se encuentra en el rango de columnas visibles, debe ponerse a 0, por lo que, si los saltos de control anteriores no se han llevado a cabo, la celda actual (direccionada por %rsi) es puesta a 0 mediante la instrucción "*movb $0, (%%rsi);*". Una vez hecho esto, se realiza el paso a la siguiente iteración, que conlleva actualizar %rsi para que pase a apuntar a la siguiente celda, incrementar la posición de columna marcada por %edi y comprobar la permanencia en el recorrido por columnas comparado la columna actual con el número total de columnas.

El segundo bloque *asm* incluye una implementación equivalente siguiendo un procesamiento vectorizado. En este caso, por simplificación, la anchura del contenedor se asume con valor de 16. Este es el valor real de la anchura, por lo que la implementación funciona correctamente, aunque no se puede utilizar para otros tamaños de contenedor. En esta segunda implementación, hay un único recorrido por filas, puesto que cada fila está compuesta por 16 celdas y es procesada de forma completa en cada iteración. Además de inicializar el registro %rsi con la dirección inicial del contenedor, antes de comenzar el bucle, el registro %rcx es inicializado con el valor del parámetro *h*, para fijar el número de iteraciones que se ejecutarán con la instrucción "*loop*". Tras estas instrucciones, el registro %xmm0 es inicializado para marcar las celdas de cada fila que se pondrán a 0 y aquellas que deberán mantener su valor. Este registro es utilizado dentro del bucle a través de una operación lógica *and* para borrar o mantener el contenido de las 16 celdas que forman cada fila. Con este objetivo, las 2 primeras y dos últimas posiciones de %xmm0 son inicializadas con el valor 0xFF y las restantes celdas con valor 0. En cuanto al bucle, en cada iteración se carga la fila actual del contenedor en el registro %xmm1, mediante la instrucción "movdqu". Debe utilizarse esta instrucción de transferencia puesto que no se puede asegurar que los datos estén alineados en memoria. A continuación, se realiza una operación *and* entre %xmm0 y %xmm1 para mantener el contenido de las 2 primeras y últimas celdas de la fila actual y poner a 0 las restantes. Tras ello, el nuevo contenido de %xmm1 es transferido al contenedor, a partir de la posición indicada por %rsi. Antes del paso a la siguiente iteración, %rsi es incrementado en 16 posiciones para que pase a direccionar la siguiente fila del contenedor.

Especificación de los objetivos de la práctica

El principal objetivo de esta práctica es completar el código de la aplicación descrita para que su funcionamiento sea el que se detalla en la sección anterior. Para ello, se deberán implementar, en lenguaje ensamblador, los procedimientos vacíos del módulo "*tetris.cpp*". Para cada uno de ellos, se proporciona la estructura inicial del bloque ensamblador, en la que se ha incluido la definición de operandos que afectan a la implementación. La lista de registros utilizados incluye únicamente la palabra "memory", ya que en todos los casos la memoria es modificada. La inclusión de registros dentro de esta lista dependerá de la implementación que se desarrolle en cada caso, por lo que, será necesario completarla para cada uno de los procedimientos.

Se describe a continuación, con más detalle, la funcionalidad de cada uno de los procedimientos a completar:

- *void* ***borraPieza****(char * **container**, int **w**, char **figure**[4][4], int **c**, int **f**)*: borra la pieza del contenedor indicada en el parámetro *figure*, situada a partir de la columna *c* y fila *f*.
- *bool* ***huecoLibre****(char * **container**, int **w**, char **figure**[4][4], int **c**, int **f**)*: comprueba si hay hueco libre en la posición del contenedor indicada por *c* (columna) y *f* (fila)

para colocar la pieza especificada en *figure*. Si existe hueco, devuelve el valor *true* y, en caso contrario, el valor *false*. Para que el valor retornado sea el adecuado, el bloque de código ensamblador debe almacenar el valor 1 (*true*) o 0 (*false*) en la variable local *hayHueco*, incluida como primer operando del bloque.

- *void* ***insertaPieza****(char * **container**, int **w**, char **figure**[4][4], int **c**, int **f**)*: inserta la pieza indicada en *figure* en la posición del contenedor especificada por *c* y *f*.
- *int* ***borraFilas****(char * **container**, int **w**, int **h**)*: borra las filas completas del contenedor. Por cada fila borrada, los bloques situados por encima, deben bajar una posición en fila. El procedimiento debe retornar el número de filas borradas. Para ello, desde el bloque de código en ensamblador, es necesario almacenar este valor en la variable *nFilas*, incluida como primer operando del bloque.
- *void* ***annadeFila****(char * **container**, int **w**, int **h**, char * **row**)*: añade la fila indicada en *row* en la parte inferior del contenedor. Antes de añadir la fila, los bloque ya situados en el contenedor suben una posición en fila.

Es obligatorio implementar el primer bloque *asm* de todos los procedimientos. La nota máxima de la parte obligatoria será de Notable(8). La implementación mediante SSE del segundo bloque *asm* de los procedimientos *borraPieza*, *huecoLibre* e *insertaPieza* será opcional. Está implementación adicional podrá suponer hasta 2 puntos más en la nota de la práctica.

Nota: la práctica se realizará de manera individual.
Fecha de entrega de la práctica: 17/01/2023 (hasta las 14:00)
Entrega: la entrega se realizará a través de la subida al aula virtual de la asignatura de un archivo .zip que contenga el proyecto completo.