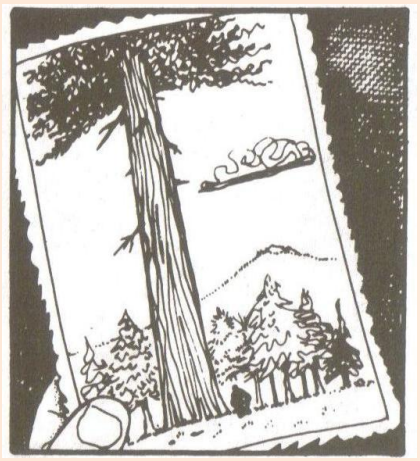

PRÁCTICA FINAL OBLIGATORIA

DE LA ASIGNATURA

INFORMÁTICA APLICADA

PARA LA TITULACIÓN DE GRADO EN

INGENIERÍA MECÁNICA (UPCT)

CURSO 2012-2013

Implementación del juego del BUSCAMINAS

Pedro María Alcover Garau

Primera versión, 26 de noviembre, 2012.
Segunda versión, 27 de noviembre de 2012.

(Todo depende de a dónde quiera llegar uno. Y es que cuando uno se pone… ¡llega!)

¡Ánimo!

INTRODUCCIÓN

El Buscaminas es un juego de ordenador para un solo jugador. El objetivo del juego es despejar un campo minado sin detonar ninguna mina.

El tablero de juego del Buscaminas es un rectángulo de filas y columnas: tantas como elija el jugador. En cada posición de fila y columna hay una celda o casilla, inicialmente cubierta. En algunas de esas celdas se ocultan minas. El jugador debe despejar todas las celdas del tablero excepto aquellas que ocultan una mina. Si el jugador da la orden de levantar una celda con mina, entonces pierde el juego.

Algunas casillas, entre las que no son celdas de mina, tienen asignado un valor numérico: ese valor indica el número de minas que circundan a la posición de esa celda. Cada celda tiene hasta un máximo de otras ocho que la rodean, y por tanto el valor máximo que podrá tomar una celda es 8.

Habrá casillas que no tendrán ninguna mina a su alrededor: a esas casillas no se les asigna número alguno (o se les asigna el valor 0). Cuando el jugador levanta una celda sin valor numérico y sin mina es inmediato saber que las ocho celdas que la rodean tampoco tienen mina: por eso, cuando el jugador selecciona una celda sin valor, el juego desvela automáticamente todas aquellas celdas que, a partir de ésa seleccionada, se encuentren también sin valor, hasta levantar todas aquellas que tienen un valor numérico distinto de 0.

Supongamos el tablero (al descubierto) que se muestra en la Figura 1. Evidentemente, el jugador desconoce la información que se recoge en esa figura: lo que realmente se le muestra es el cuadro de la Figura 2.

	c00	c01	c02	c03	c04	c05	c06	c07	c08	c09
f00	0	0	0	1	M	2	2	2	1	0
f01	1	1	1	1	2	3	M	M	2	0
f02	1	M	2	1	1	M	4	M	2	0
f03	1	2	M	1	1	1	2	1	1	0
f04	0	1	1	1	0	0	0	0	0	0
f05	0	1	1	1	0	0	0	0	1	1
f06	1	2	M	2	1	0	0	0	1	M
f07	1	M	3	M	1	0	0	0	1	1
f08	1	1	2	1	1	0	0	0	0	0
f09	0	0	0	0	0	0	0	0	0	0

Figura 1.: Tablero (10 x 10 al descubierto) de una partida del Buscaminas.

	c00	c01	c02	c03	c04	c05	c06	c07	c08	c09
f00	-	-	-	-	-	-	-	-	-	-
f01	-	-	-	-	-	-	-	-	-	-
f02	-	-	-	-	-	-	-	-	-	-
f03	-	-	-	-	-	-	-	-	-	-
f04	-	-	-	-	-	-	-	-	-	-
f05	-	-	-	-	-	-	-	-	-	-
f06	-	-	-	-	-	-	-	-	-	-
f07	-	-	-	-	-	-	-	-	-	-
f08	-	-	-	-	-	-	-	-	-	-
f09	-	-	-	-	-	-	-	-	-	-

Figura 2.: Tablero (10 x 10) de inicio para el juego del Buscaminas.

Supongamos que el usuario decide destapar la celda que está en la fila 0 y la columna 3. Le indicará al ordenador esos dos parámetros (de fila y columna) y le indicará que lo que desea hacer con esa celda es levantarla. Entonces, el tablero adoptará, en esa zona, el aspecto de la Figura 3: mostrará el valor 1 de esa posición levantada, indicando que en torno a esa posición hay una mina y sólo una.

	c00	c01	c02	c03	c04	c05
f00	-	-	-	1	-	-
f01	-	-	-	-	-	-

Figura 3.: Aspecto parcial del Tablero del Buscaminas cuando se ha levantado la celda en posición fila 0 y columna 3. El valor 1 indica que en torno a esta posición hay una mina (y sólo una). Ésta puede estar localizarla en la fila 0, columnas 2, ó 4; o en la fila 1, columnas 2, 3, ó 4.

Supongamos ahora que el usuario selecciona la posición fila 9 y columna 9. Como se puede ver en la Figura 1, esa celda tiene el valor 0 (evidentemente, esa información que nosotros ahora conocemos, no está disponible para el jugador). Al seleccionar, pues, el jugador, esa celda, el tablero deberá presentar la apariencia mostrada en Figura 4.

Desde luego, el inicio del juego queda sujeto al azar. Pudiera ser que el usuario hubiera escogido inicialmente la celda de fila 0 y columna 4 (ver en Figura 1): en ese caso, hubiera "explosionado" la mina y el juego hubiera terminado y el jugador habría perdido la partida. Pero una vez el juego ha comenzado quedan pocos flecos para el azar, y casi todas la celdas van quedando claras aunque no se hayan descubierto. Por ejemplo, en la

Figura 4 es fácil darse cuenta que tenemos una mina en las posiciones fila 2, columna 7; ó fila 7, columna 3; o fila 6, columna 9. Entonces el jugador puede seleccionar esas posiciones como mina, para recordarle dos cosas: (1) que esas posiciones no deben ser levantadas; y (2) que si allí hay una mina, en todas las celdas lindantes con ella y que tengan ya descubierto el valor 1 podemos ya saber que todas las demás celdas que las rodeen están limpias de minas. En el Cuadro 5 se muestran esas tres celdas señaladas como minas.

	c00	c01	c02	c03	c04	c05	c06	c07	c08	c09
f00	-	-	-	1	-	-	-	-	1	
f01	-	-	-	-	-	-	-	-	2	
f02	-	-	-	-	-	-	-	-	2	
f03	-	-	-	1	1	1	2	1	1	
f04	-	-	-	1						
f05	-	-	-	1					1	1
f06	-	-	-	2	1				1	-
f07	-	-	-	-	1				1	1
f08	1	1	2	1	1					
f09										

Figura 4.: Aspecto que adquiere el Tablero cuando se he seleccionado la Fila y Columna 9, donde teníamos un valor 0. Como se explica en esta memoria, al levantar una celda con valor 0, se debe proceder a levantar, de forma automática, todas sus celdas adyacentes. Si alguna de sus adyacentes vuelve a tener el valor 0, entonces, de nuevo, el programa deberá proceder a levantar todas sus celdas adyacentes.

Puede ocurrir que el jugador marque una celda como mina pero que, en realidad, no sea tal. El juego, en ese caso, deberá seguir: el programa marcará esa celda como mina, pero no terminará mientras que el jugador se desdiga y marque esa celda, finalmente, como celda libre de minas. De todas formas, si el jugador marca equivocadamente una celda como celda con mina, es muy posible que pierda el juego, porque en algún momento no le saldrán las cuentas de las celdas numeradas al descubierto, y pensará que no hay mina donde realmente sí la había.

Desde luego, el entorno más cómodo para desarrollar el juego del Buscaminas sería con una herramienta gráfica, donde el usuario seleccionara las celdas como limpias o como posibles minas pulsando sobre ellas el botón izquierdo o derecho del ratón. Desde luego eso no se puede hacer en este curso donde los alumnos no conocen cómo desarrollar la parte gráfica del software. Pero la algoritmia en cualquier caso es la misma, y el programa puede implementarse para ser ejecutado en una ventada de consola. Así se les propone

que lo hagan. Además, se les pide que implementen la aplicación de tal manera que el tablero pueda tener cualquier tamaño de filas y columnas.

	c00	c01	c02	c03	c04	c05	c06	c07	c08	c09
f00	-	-	-	1	-	-	-	-	1	
f01	-	-	-	-	-	-	-	-	2	
f02	-	-	-	-	-	-	-	<?>	2	
f03	-	-	-	1	1	1	2	1	1	
f04	-	-	-	1						
f05	-	-	-	1					1	1
f06	-	-	-	2	1				1	<?>
f07	-	-	-	<?>	1				1	1
f08	1	1	2	1	1					
f09										

Figura 5.: Aspecto del tablero cuando se han marcado tres celdas con el indicativo de que, en ellas se halla una mina. Esta marca no garantiza que, de hecho, haya una mina en esa posición. Es una marca que puede poner el jugador, porque él piensa que en esa celda hay una mina. Si está en lo cierto, entonces esa marca le será de gran ayuda. Si estaba equivocado, es muy probable que esta marca le lleve a considerar que no hay mina en una posición que, de hecho, sí la hay.

CÓMO IMPLEMENTAR ESTA APLICACIÓN

TIPOS DE DATO

La primera cuestión a decidir es la de la forma en que se va a codificar la información del tablero: porque en cada una de sus celdas deberemos consignar dos informaciones:

(1) por un lado, su contenido: la información sobre si en esa celda hay o no hay una mina; y, si no la hay, el valor numérico calculado para esa celda en función del número de minas que la rodean;

(2) por otro lado deberemos indicar sobre cada celda su estado: oculta, levantada, marcada como mina, o mina reventada, lo que lleva también consigo el fin del juego.

Para lograr que una celda mantenga dos informaciones tenemos algunas soluciones. La más sencilla es crear una estructura de datos, es decir, un nuevo tipo de dato. Podría tener la siguiente forma:

```
typedef struct
{
      short int valor_celda;
      short int estado_celda;
}Celda;
```

Así hemos creado un nuevo tipo de dato, llamado `Celda`. Con este tipo de dato, podemos hacer referencia a uno u otro campo, mediante el operador punto. Veámoslo con un ejemplo. Si creamos una variable de tipo `Celda`:

```
Celda mi_celda;
```

Ahora podemos hacer referencia a uno u otro valor de esa variable. Lo hacemos de la siguiente manera:

- `mi_celda.valor_celda`, para referirnos al primer campo;
- `mi_celda.estado_celda`, para referirnos al segundo campo.

Esta guía trabajará con el tipo de dato `Celda`. No hay ningún problema en que cualquier alumno decida trabajar la información de otra manera (más adelante se muestra cómo). Quizá con lo explicado en esta página sea suficiente para comprender cómo se manejan esos nuevos tipos de dato. Si desea conocer mejor la creación y el manejo de las estructuras, entonces quizá sea conveniente que trabaje unas pocas páginas del capítulo 20 del manual de Teoría.

Así pues, la declaración de la matriz del tablero podría tener el siguiente aspecto:

```
#define _FILAS                          10
#define _COLUMNAS                       10
Celda tablero[_FILAS][_COLUMNAS];
```

Donde la dimensión puede fijarse mediante directivas define, o se puede solicitar al usuario que, al inicio del juego, decida el tamaño que debe tener el tablero sobre el que va a jugar. En esta presentación, consideraremos todo el tiempo que el programa se realiza sobre los valores de filas y columnas definidas en estas directivas. No supone complicación alguna sobre lo que aquí se dirá el hecho de que esos dos valores (`_FILAS` y `_COLUMNAS`) sean dos variables cuyo valor introduce el usuario en tiempo de ejecución.

Otra forma de crear el tablero para el Buscaminas es con un array de tres dimensiones. Sería como crear dos matrices iguales en tamaño (en filas y columnas): una de ellas dedicada a los valores del tablero, y la otra dedicada a codificar el estado de cada una de las posiciones. Sería algo así:

```
#define _FILAS                          10
#define _COLUMNAS                       10
short int tablero[2][_FILAS][_COLUMNAS];
```

Así, cuando quisiéramos hacer referencia al valor de la celda, trabajaríamos con la matriz `tablero[0][i][j]`, para valores de `i` entre 0 y `_FILAS` y de `j` entre 0 y `_COLUMNAS`. Y cuando quisiéramos hacer referencia al estado de la celda, trabajaríamos con la matriz `tablero[i][i][j]`, con los mismos posibles valores para los índices `i` y `j`.

Aunque esta memoria está presentada de acuerdo al primer esquema de codificación del tablero (mediante la estructura `Celda`) bien puede interpretarse toda la información de acuerdo con esta segunda forma. Baste tener en cuenta que, de acuerdo con estas dos formas de codificar el tablero del Buscaminas, tenemos que decir `tablero[i][j].valor_celda` es equivalente a decir `tablero[0][i][j]`, y que decir `tablero[i][j].estado_celda` es equivalente a decir `tablero[1][i][j]`.

En los folios que siguen se le ofrece una guía para que le ayude a resolver la práctica. Decida ya cuál de los dos esquemas de codificación del tablero va a elegir, y no cambie su decisión a mitad de trabajo.

Hay otras diversas formas de codificar el tablero. Algunas lograrían un tablero mucho más reducido en la memoria, con el que trabajaríamos mediante operadores a nivel de bit. Es una filosofía sencilla, pero requiere oficio en el uso de estos operadores. No las vamos a mostrar ahora aquí, para evitar el desconcierto.

VALORES PREDEFINIDOS

Otra Cuestión es la colección de valores que daremos a las celdas en el campo de control. Es decir, una vez tenemos creado el tablero… ¿Qué valores pueden tomar cada una de las 100 celdas, en el campo correspondiente a `tablero[f][c].estado_celda` (o `tablero[1][i][j]`? Porque esa variable, de tipo **`short`**, no tiene sentido que pueda tomar cualquier valor, si en realidad hemos definido sólo cuatro valores posibles, que son: (1) celda oculta; (2) celda levantada; (3) celda marcada como mina; o (4) celda con mina reventada. Lo más sencillo sería crear un nuevo tipo de dato enumerado, pero no hemos visto esto en clase de teoría (Capítulo 20 del manual). Vamos a proponer una solución que no exige presentar más materia que no hemos presentado en las clases teóricas de la asignatura.

Podemos crear una serie de valores con directivas define:

```
#define _OCULTA                 1
#define _LEVANTADA              2
#define _MINA_MARCADA           3
#define _MINA_REVENTADA         4
```

Respecto a los valores que pueden adoptar en las distintas posiciones del array el campo `tablero[f][c].valor_celda` (ó también `tablero[0][i][j]`) ya hemos dicho que éstos pueden ser cualquiera entre 0 y 8, ambos inclusive. Pero también debemos indicar qué código dar a una celda que tenga una mina; en ese caso la información que debemos consignar en ella no es cuántas minas encuentra a su alrededor, sino simplemente que esa celda corresponde a una mina. Podemos, por ello, crear un valor (distinto de cualquier entero entre 0 y 8) que signifique que estamos ante una mina. Por ejemplo:

```
#define _MINA                   0x7FFF
```

Respecto a las sucesivas selecciones que el jugador irá haciendo, es necesario conocer si la selección de una nueva celda se realiza para que ésta quede destapada, o si lo que pretende el jugador es marcar esa celda como posible mina. Así, cada nueva entrada del jugador, deberá recoger tres valores: los dos primeros indicarán la fila y la columna de la celda seleccionada; el tercero deberá indicar el motivo por el que el jugador ha seleccionado la celda. Convendrá crear otros dos valores para esa opción de selección:

```
#define _SELECCION_LEVANTAR_CELDA       1
#define _SELECCION_MARCAR_MINA          2
```

Con todo esto, ya tenemos creados los tipos de dato y también hemos acotado los valores posibles que puede adoptar una celda del tablero del buscaminas. Más adelante, en esta memoria, podrá encontrar algunos nuevos literales creados para la gestión de la información en la aplicación: ya los verá. Podemos ahora comenzar a implementar el código.

FUNCIONES

Para resolver este trabajo se sugiere que se declaren y se definan una serie de funciones. Entre ellas, enumeramos tres a continuación, en Código 1.

Código 1.: Declaración de algunas funciones, posiblemente necesarias para resolver esta aplicación, de las que vamos a mostrar, en esta memoria, una posible implementación. Se presentan los prototipos de las funciones según cada una de las dos formas de codificar el tablero que hemos propuesto anteriormente.

```
void inicializarTablero
      (short fil, short col, Celda tablero[fil][col]);

void pintarTablero
      (short fil, short col, Celda tablero[fil][col]);

void eliminarCerosAdyacentes
      (short f, short c, short fil, short col,
                  Celda tablero[fil][col]);
```

```
void inicializarTablero
      (short fil, short col, short tablero[2][fil][col]);

void pintarTablero
      (short fil, short col, short tablero[2][fil][col]);

void eliminarCerosAdyacentes
      (short f, short c, short fil, short col,
                  short tablero[2][fil][col]);
```

FUNCIÓN `inicializarTablero()`

La primera de estas funciones inicializa el cuadro del tablero. Además de decidir cuántas minas se van a insertar en el tablero, deberá hacer otras dos cosas:

(1) Elegir, de forma aleatoria algunas de las celdas y asignarles el papel de minas dentro del juego. Evidentemente, la función deberá vigilar que no se inserte dos veces una mina en una misma celda.

(2) Recalcular, ante cada nueva mina insertada, el valor de todas las celdas que circundan a la celda donde se ha insertado la nueva mina. Para ello se deberá tener en cuenta que:

   a. Las celdas del cuadro ubicadas en los bordes del tablero no están rodeadas por 8 celdas sino por algunas menos: 5 ó 3, según esa celda sea lateral o uno de los cuatro vértices. Al hacer el recuento se deberá, pues, vigilar para no hacer violación de memoria, y no se intente asignar valores a hipotéticas posiciones del tablero con índices negativos o mayores que el tamaño de cada dimensión.

   b. Las celdas marcadas como mina no deben sufrir modificación alguna.

Una posible implementación de esta primera función podría ser la que se propone en Código 2.

Código 2.: Implementación de la función `inicializarTablero`, que asigna valores iniciales al tablero del buscaminas. Mostramos sólo la implementación para la codificación del tablero mediante el tipo de dato matriz de dos dimensiones de tipo Celda. Para el caso de la matriz tridimensional de tipo **`short`**, bastaría, como ya se explicó previamente, cambiar cualquier referencia a `tablero[f][c].estado_celda` por `tablero[1][f][c]`; y cualquier referencia a `tablero[f][c].valor_celda` por `tablero[0][f][c]`.

```
#include <stdlib.h>          // para la generación de aleatorios.
#include <time.h>            // para la función time()

// Definición ...
void inicializarTablero(short fil, short col, Celda tablero[fil][col])
{
// fil: número de filas del tablero.
// col: número de columnas del tablero.
    short numeroMinas;       // minas a insertar en el tablero
    short contadorMinas;     // contador de las minas ya insertadas
    short f, c;              // fila y columna actual en cada momento.

    // Inicializamos la semilla del generador de pseudoaleatorios.

    srand(time(NULL));

    // Determinamos de forma aleatoria el número de minas que se
    // van a insertar en el tablero: un mínimo de 10 y algunas más
    // entre 1 y un 5% del producto de filas por columnas.

    numeroMinas = 10 + (rand() % (fil * col) / 20 + 1);

    // Inicializamos el tablero: Todas las celdas ocultas
    // y con valor igual a 0.
```

Código 2.: Función `inicializarTablero` (Cont).

```
    for(f = 0 ; f < fil ; f++)
    {
        for(c = 0 ; c < col ; c++)
        {
            tablero[f][c].estado_celda = _OCULTA;
            tablero[f][c].valor_celda = 0;
        }
    }

    // Comienza la inserción de minas

    for(contadorMinas = 0 ; contadorMinas < numeroMinas ; )
    { // inicio bloque 1
        f = rand() % fil;     // fila aleatoria.
        c = rand() % col;     // columna aleatoria.

    // Debemos verificar que en esa posición no hay, ya, una mina
    // Si no hay mina habrá que realizar dos operaciones:
        if(tablero[f][c].valor_celda != _MINA)
        { // inicio bloque 2
            short ff, cc;

        // (1) Inserción de una nueva mina

            tablero[f][c].valor_celda = _MINA;
            contadorMinas++;  // Una nueva mina insertada

        // (2) Incrementar en 1 todas las celdas adyacentes.

            for(ff = f - 1 ; ff <= f + 1 ; ff++)
            { // inicio bloque 3
                for(cc = c - 1 ; cc <= c + 1 ; cc++)
                { // inicio bloque 4

    // Verificamos que esa celda existe
                    if((ff >= 0)   &&
                       (cc >= 0)   &&
                       (ff < fil)  &&
                       (cc < col))
                    { // inicio bloque 5
    // Verificamos, en cada celda adyacente, que NO hay ya una mina
                        if(tablero[ff][cc].valor_celda != _MINA)
                        { // inicio bloque 6
                            tablero[ff][cc].valor_celda++;
                        } // fin bloque 6
                    } // fin bloque 5
                }  // fin bloque 4
```

Código 2.: Función `inicializarTablero` (Cont).

```
            } // fin bloque 3
        } // fin bloque 2
    } // fin bloque 1

    return;
}
```

FUNCIÓN `pintarTablero()`

La segunda función se encarga de mostrar el tablero por pantalla. Aquí proponemos una versión que imprimirá en pantalla de comando un cuadro similar al mostrado en la Figura 2. En Código 3 puede ver una posible implementación. Quizá le convendrá crear alguna otra función que imprima en tablero en algunos casos especiales (o modificar ligeramente ésta): cuando el jugador haya logrado ganar la partida, y se pueda entonces mostrar ya el cuadro con toda la información, tal y como se muestra en la Figura 1.

Código 3.: Implementación de la función `pintarTablero`, que muestra por pantalla el tablero del buscaminas. De nuevo, recuerde que si usted ha elegido definir el tablero como un array tridimensional, únicamente bastará con que cambie cualquier referencia a `tablero[f][c].estado_celda` por `tablero[1][f][c]`; y cualquier referencia a `tablero[f][c].valor_celda` por `tablero[0][f][c]`. La declaración del array en el parámetro de entrada, como ya se vio en la declaración del prototipo, se obtiene al cambiar `Celda tablero[fil][col]` por **`short`** `tablero[2][fil][col]`.

```
#include <stdlib.h>          // Para la función system().
#include <stdio.h>           // Para la función printf().

// Definición ...
void pintarTablero(short fil, short col, Celda tablero[fil][col])
{
    short f, c;

// Limpiamos la pantalla. Para ello invocamos a la función system,
// que ejecuta una orden desde el sistema operativo.
// La orden "cls" limpia la pantalla de comandos en Windows.
// La orden “clear” limpia la pantalla de comandos en Linux y Mac.
```

Código 3. Función `pintarTablero` (Cont.).

```
    system("cls");
//  system("clear");
    printf("\n\t\t");

// Mostramos por pantalla la numeración de cada columna...

    for(c = 0 ; c < col ; c++)
    {
        printf(" c%02hd ", c);
    }

// Mostramos ahora, una a una, todas las filas...

    for(f = 0 ; f < fil ; f++)
    { // inicio bloque 1
        // Primero el número de la fila ...
        printf("\n\n\tf%02hd\t", f);

        // Ahora todas las columnas de la fila actual ...

        for(c = 0 ; c < col ; c++)
        { // inicio bloque 2
          // Si la celda está oculta, no la mostramos.

            if(tablero[f][c].estado_celda == _OCULTA)
            { // inicio bloque 3
                printf("  -  ");
            } // fin bloque 3.
          // Si la celda está levantada, la mostramos...
          // Puede ser que tengamos que mostrar una MINA.
          // O puede ser que tengamos que mostrar un valor numérico.

            else if(tablero[f][c].estado_celda == _LEVANTADA)
            { // inicio bloque 3
              // Si hay una mina (opción imposible, pero ahí va):

                if(tablero[f][c].valor_celda == _MINA)
                { // inicio bloque 4
                    printf("  M  ");
                } // fin bloque 4

              // Hay un valor numérico distinto de cero.

                else if(tablero[f][c].valor_celda != 0)
                { // inicio bloque 4
                    printf("  %hd  ", tablero[f][c].valor_celda);
                } // fin bloque 4
```

Código 3. Función `pintarTablero` (Cont.).

```
            // Hay un valor numérico igual a cero.
              else
              { // inicio bloque 4
                  printf("     ");
              } // fin bloque 4
          } // fin bloque 3

        // Si la celda está marcada como mina, mostramos una señal...

          else if(tablero[f][c].estado_celda == _MINA_MARCADA)
          { // inicio bloque 3
              printf(" <?> ");
          } // Fin bloque 3

        // Si la celda es una mina que hemos "pisado"...

          else if(tablero[f][c].estado_celda == _MINA_REVENTADA)
          { // inicio bloque 3
              printf(">EXP<");
          } // fin bloque 3
      } // fin bloque 2
  } // fin bloque 1

  printf("\n\n\n");

  return;
}
```

FUNCIÓN `eliminarCerosAdyacentes()`

La tercera función la hemos llamado `eliminarCerosAdyacentes`, y se encarga de levantar todas las celdas que estén en torno a una de valor cero que haya sido levantada previamente. Ya se ha explicado en la introducción el porqué de esta operación. En Código 4 puede ver una posible implementación de esta función. En este caso, ofrecemos una solución **recursiva** (cfr. Capítulo 15 del manual de referencia de teoría): sería útil que comprendiera bien este código.

Con éstas, tiene ya tres funciones necesarias dentro de la aplicación que deseamos implementar. No se le va a mostrar aquí más código (bueno: en realidad algo más le facilito más adelante. Ya lo verá). Le corresponde ahora a usted continuar con el desarrollo de nuestro proyecto. Se le facilitan, sin embargo, algunas orientaciones. Posiblemente, con este código aquí mostrado, tiene usted ya resuelta más del 50% de la

aplicación: la correspondiente a la parte más complicada de la implementación. Haga un esfuerzo por entender estas implementaciones aquí mostradas: le darán muchas pistas sobre cómo resolver todo lo que falta; y le ayudará a aprender a programar.

Código 4.: Implementación recursiva de la función `eliminarCerosAdyacentes`, que levanta todas las celdas de valor cero, adyacentes a una seleccionada y de valor cero.

Por tercera y última vez, recuerde que si usted ha elegido definir el tablero como un array tridimensional, únicamente bastará con que cambie cualquier referencia a `tablero[f][c].estado_celda` por `tablero[1][f][c]`; y cualquier referencia a `tablero[f][c].valor_celda` por `tablero[0][f][c]`. La declaración del array en el parámetro de entrada, como ya se vio en la declaración del prototipo, se obtiene al cambiar `Celda tablero[fil][col]` por **`short`** `tablero[2][fil][col]`.

```
// Esta función es invocada cuando la celda marcada
// por la fila f y la columna c tiene el valor 0.

// Definición ...
void eliminarCerosAdyacentes
(short f, short c, short fil, short col, Celda tablero[fil][col])
{
// f, c: fila y columna de la celda de valor cero con la que trabajamos.
// fil, col: dimensiones del tablero.
    short ff, cc;     // para recorrer las filas y columnas del tablero.

// Levantamos la celda seleccionada, (indicada por f y c).
// Esta operación es importante para evitar concurrencia infinita:

    tablero[f][c].estado_celda = _LEVANTADA;

// Y ahora levantamos sus adyacentes, todas ellas sin mina.
// Además, si en estas adyacentes encontramos una celda con valor 0
// de nuevo invocamos la función para levantar todas las de su
// alrededor.

    for(ff = f - 1 ; ff <= f + 1 ; ff++)
    { // inicio bloque 1
        for(cc = c - 1 ; cc <= c + 1 ; cc++)
        { // inicio bloque 2
         // Verificamos que la celda existe:

            if((ff >= 0) && (cc >= 0) &&
               (ff < fil) && (cc < col) &&
               ((ff != f) || (cc != c)))
            { // inicio bloque 3
             // Si la celda ya ha sido levantada antes, entonces no
             // que volver a procesarla.
```

Código 4.: Función `eliminarCerosAdyacentes` (Cont.).

```
                if(tablero[ff][cc].estado_celda != _LEVANTADA)
                { // inicio bloque 4
                 // Levantamos la celda

                    tablero[ff][cc].estado_celda = _LEVANTADA;

                 // Y si la celda vale 0, volvemos a invocar
                 // a la función de forma recursiva o recurrente.

                    if(tablero[ff][cc].valor_celda == 0)
                    { inicio bloque 5
                        eliminarCerosAdyacentes
                             (ff, cc, fil, col, tablero);
                    } // fin bloque 5
                } // fin bloque 4
            } // fin bloque 3
        } // fin bloque 2
    } // fin bloque 1

    return;
}
```

ARCHIVOS DE CÓDIGO

Le recomendamos que, dentro del proyecto que se va a crear y donde se recogerá todo el código de la aplicación, cree 3 archivos:

- **`funciones.h`**: donde se recogerán todas las definiciones de tipo de dato, y todas las directivas define. También deberán recogerse todas las declaraciones de todas las funciones.
- **`funciones.c`**: donde se recogen todas las implementaciones de las funciones declaradas en `funciones.h`. Este archivo deberá tener, al inicio, la siguiente directiva:

  ```
  #include "funciones.h"
  ```

  Y también aquellas inclusiones de archivos de cabecera que se necesiten para el uso de las funciones en ese archivo.
- **`main.c`**: donde se recoge la función principal. Este archivo también deberá recoger la inclusión del archivo de cabecera `funciones.h`.

SUGERENCIAS SOBRE LA IMPLEMENTACIÓN DE OTRAS FUNCIONES

Desde luego, necesita más funciones, además de aquellas tres que se le han sugerido en un apartado anterior. Hay otras dos que son necesarias para la comunicación con el usuario.

(1) Una encargada de recoger cada nueva entrada que ejecuta el usuario del juego implementado. Cada entrada debe estar formada por tres valores: uno que indique la fila de la nueva celda seleccionada; otro que indique la columna de esa nueva celda seleccionada; y un tercer valor que indique si el usuario elige esa celda para ser destapada o para ser marcada como mina. Esa función podría tener, por ejemplo, la declaración sugerida en Código 5.

Código 5.: Declaración de la función `nuevaEntrada()`. Se muestra la declaración según cada una de las dos formas que hemos propuesto para la creación del tablero del Buscaminas.

```
short nuevaEntrada
   (short*f , short*c, short fil , short col ,
    Celda tablero[fil][col]);
```

```
short nuevaEntrada
   (short*f , short*c, short fil , short col ,
    short tablero[2][fil][col]);
```

Esta función devolverá el motivo de la selección de una celda determinada: o para que ésta sea levantada, o para que ésta sea marcada como posible mina. Pero, ¿dónde recogerá la función el valor de la fila y de la columna seleccionada?: pues en los parámetros `f` y `c`, primero y segundo de la función y que aquí se pasan ¡por referencia! (cfr. Capítulos 16 y 17 del manual de referencia de teoría, o Capítulo 10 del manual de prácticas). Esta función tiene una definición muy sencilla: debe únicamente recoger esos tres valores: dos de ellos deberá dejarlos asignados en las variables `f` y `c`, y el tercero (`_SELECCION_MARCAR_MINA` ó `_SELECCION_LEVANTAR_CELDA`), deberá devolverlo con la sentencia **`return`**.

(2) Otra función deberá encargarse de analizar la entrada recibida. Es decir, deberá buscar la celda seleccionada (operación que se habrá realizado con la función anterior) y deberá decidir qué valor dar al estado de la celda:

a. Si la celda ha sido seleccionada con la opción `_SELECCION_MARCAR_MINA`, entonces se le deberá dar el estado de la celda el valor `_MINA_MARCADA`.

b. Si la celda ha sido seleccionada con la opción `_SELECCION_LEVANTAR_CELDA`, entonces se deberá dar al estado de la celda uno de los siguientes valores:

   i. Si la celda es una mina, entonces el estado será `_MINA_REVENTADA`. En este caso, nuestra función deberá devolver un valor que indique que la partida ha terminado y que el jugador ha perdido.

   ii. Si la celda es un espacio con un valor numérico distinto de cero, entonces el estado deberá ser `_LEVANTADA`. En este caso, nuestra función deberá devolver un valor que indique que la partida sigue.

   iii. Si la celda es un espacio con el valor numérico cero, entonces deberá invocar a la función `eliminarCerosAdyacentes()`, que ya ha sido presentada.

Además, la función deberá determinar si ya se han logrado levantar todas las celdas (excepto, claro está, las que corresponden a cada una de las minas), y en ese caso, deberá devolver un valor que indique que el juego ha terminado. O deberá devolver un valor que indique que el juego continúa, si aún no se ha finalizado. O deberá devolver un valor que indique juego perdido, si el jugador ha levantado una celda que contenía una mina. De nuevo, pues, se hace conveniente crear una serie de valores que serán los de retorno de esta función:

```
#define _EXPLOSION                  1
#define _JUEGO_TERMINADO            2
#define _JUEGO_EN_MARCHA            3
```

UNA NUEVA AYUDA

Como posiblemente vaya perdido, se le ofrece una nueva ayuda, y se le facilita una posible implementación de la función `analizarEntrada` antes presentada. Puede verla en Código 6.

Código 6.: Función `analizarEntrada()`.

Recuerde (ya se ha repetido hasta la saciedad) cómo cambiar este código en el caso de que haya elegido usted trabajar con un tablero en forma de matriz tridimensional.

```
// Función que recibe como parámetros la opción seleccionada
// por el jugador (levantar o marcar celda), la fila y la columna
// de la celda seleccionada, y las dimensiones
// de la matriz del tablero así como el tablero.

short analizarEntrada
      (short op, short f, short c, short fil, short col,
       Celda tablero[fil][col])
{
      short ff, cc;

      switch(op)
      {
      case _SELECCION_MARCAR_MINA:
            tablero[f][c].estado_celda = _MINA_MARCADA;
            break;

      case _SELECCION_LEVANTAR_CELDA:
            if(tablero[f][c].valor_celda == 0)
            {
                  eliminarCerosAdyacentes(f, c, fil, col, tablero);
            }

            else if(tablero[f][c].valor_celda == _MINA)
            {
                  tablero[f][c].estado_celda = _MINA_REVENTADA;
                  return _EXPLOSION;
            }

            else
            {
                  tablero[f][c].estado_celda = _LEVANTADA;
            }
      }
```

Código 4.: Función `analizarEntrada` (Cont.).

```
      for(ff = 0 ; ff < fil ; ff++)
      {
            for(cc = 0 ; cc < col ; cc++)
            {
                  if(tablero[ff][cc].valor_celda != _MINA &&
                     tablero[ff][cc].estado_celda != _LEVANTADA)
                  {
                        return _JUEGO_EN_MARCHA;
                  }
            }
      }
      return _JUEGO_TERMINADO;
}
```

INDICACIONES FINALES

Se le ha facilitado mucha ayuda: se le ha explicado paso a paso qué debe hacer el código que usted deberá implementar. Se le ha facilitado mucha parte de este código.

Pero no tiene todo el trabajo hecho. Deberá comprender, si de verdad quieren emplearlo, el código que se le facilita. Deberá ser capaz de ensamblarlo, y lograr montar felizmente un código completo que ofrezca al usuario la posibilidad de jugar al cuadro mágico.

Y le queda, cómo no, implementar la función `main`.

Desde luego, puede afrontar el trabajo propuesto sin hacer uso alguno de lo que en esta guía se les dice. Siempre hay alumnos que así lo hacen, y siempre logro ver, gracias a ellos, soluciones originales (a mí me lo parecían).

No se desanimen ante el enunciado de la práctica. Que sea largo no es un argumento contra él, sino más bien a favor: si fuera más largo, quizá sería porque aún les facilitaba más código. Pero no puedo darles más, porque poco es lo que falta para tenerlo terminado.

Insisto: si no comprenden el código que se les facilita, no podrán saber cómo ensamblarlo. Y tampoco sabrán qué funciones nuevas harían falta para completar la implementación. Tampoco pueden adivinar, sin conocimientos, el diseño de la función `main`.

No deje este trabajo para “mas tarde”. No decida que es demasiado difícil para usted. Fundamentalmente porque no es cierto.

¡Ánimo!