Buscaminas 2.0

Fundamentos de la Programación II

Grados de la Facultad de Informática (UCM)

Fecha máxima de entrega: 29/04/2025 a las 9:00

Normas de realización de la práctica

1. La fecha límite para entregar la práctica es el **29/04/2025 a las 9:00**.
2. Debe entregarse a través del Campus Virtual, en la actividad Entrega práctica (versión 2).
3. Sigue los pasos que aparecen en el enunciado y ten en cuenta todas las indicaciones.
4. Aunque recomendamos que vayas haciendo la práctica de manera incremental, tal y como aparece en el enunciado, debes leerlo completamente antes de empezar.
5. En el Campus Virtual encontrarás algunos archivos para realizar pruebas. También encontrarás el archivo `"checkML.h"`.
6. No puedes añadir nuevos métodos públicos. Si está permitido añadir tantos métodos privados como consideres necesario.

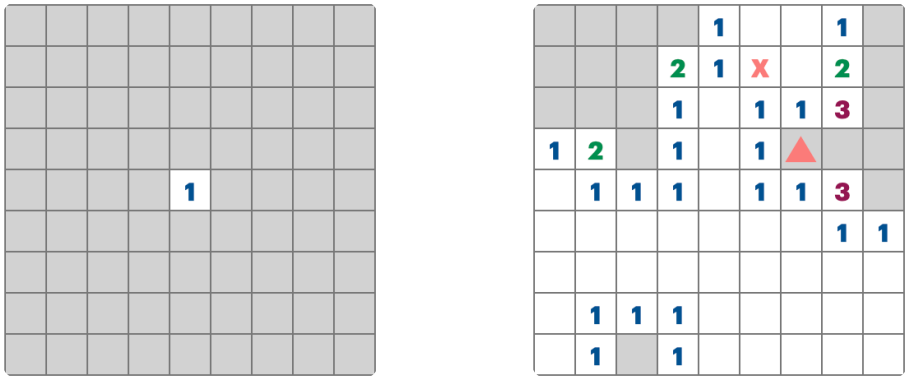
Primero resuelve el problema. Entonces, escribe el código.

— John Johnson

Añadiendo recursión

En esta versión del Buscaminas vamos a introducir nuevas funcionalidades que debes implementar de forma progresiva. En primer lugar, una vez elegida una casilla, vamos a aplicar las reglas del juego original. Es decir, además de descubrir las ocho celdas vecinas cuando la celda seleccionada está *vacía*, este proceso se repetirá, recursivamente, para todas las celdas vecinas que estén *vacías*.

En la siguiente figura, a la izquierda aparece un tablero donde la celda descubierta está rodeada por una mina, por lo tanto no se descubre ninguna celda vecina. En la figura de la derecha, la celda elegida es la marcada con "**X**". En este caso la celda está vacía y se descubren sus ocho celdas vecinas. De las ocho vecinas, para aquellas que están vacías, se descubren también sus ocho celdas vecinas y así sucesivamente hasta que no se puedan descubrir más. Para seleccionar la siguiente celda se obtiene información de las celdas descubiertas. Por ejemplo, se sabe que la celda marcada con "▲" contiene una mina con toda seguridad, por lo tanto no debe elegirse.



Para implementar esta forma de descubrir celdas simplemente debes cambiar la implementación de la función `juega(fila,columna,lista_posiciones)` de la clase `Juego`. Dada la naturaleza recursiva del proceso de descubrir celdas, debes implementar la función de forma recursiva.

Incorporando gestión de memoria dinámica y ordenación

Vamos a incorporar memoria dinámica a la práctica mediante el uso de arrays dinámicos, arrays estáticos de punteros y arrays dinámicos de punteros. Recuerda incluir el fichero `checkML.h` en aquellos módulos donde gestiones la memoria. Implementa los siguientes apartados en orden, y no realices el siguiente hasta haber comprobando que el anterior funciona correctamente y no deja memoria utilizada sin borrar.

Usando un array dinámico para implementar la lista de posiciones

En la versión anterior, la clase `ListaPosiciones` estaba implementada con un array de tamaño fijo. En esta nueva versión utilizaremos un array dinámico para almacenar las posiciones. Adapta la implementación de la clase para el atributo `Posicion* lista`, y recuerda que debes redimensionar el array cuando se quede sin espacio. Puesto que debes incorporar una

destructora a la clase, el método `destruye()`, que hacía las veces de destructora en la versión 1, debes eliminarlo o ponerlo privado (cambiando la implementación para que pueda usarla la destructora). Concretamente debes añadir a esta clase:

```
ListaPosiciones(const ListaPosiciones& lp);
~ListaPosiciones();
```

donde la constructora por copia hace una copia de lp en `this`, y la destructora libera la memoria ocupada por el array dinámico.

ListaUndo como array estático de punteros

En esta clase vamos a cambiar el atributo lista por `ListaPosicones* lista[MAX_UNDO]`. El funcionamiento de la lista debe ser igual que en la versión 1. De nuevo tienes que eliminar la función `destruye()` o ponerla privada. Necesitarás además implementar la destructora (que debe liberar la memoria que se haya creado en esta clase), modificar el método `ultimo_elemento()`, que ahora no decrementa el contador de la lista, y añadir el método:

```
void eliminar_ultimo();
```

que elimina el último elemento de la lista y decrementa el contador de la lista en 1.

Debes tener en cuenta que al insertar un elemento tienes que crear memoria nueva y utilizar la constructora por copia de la clase `ListaPosiciones`, para evitar así comparticiones de memoria que pueden generar errores en ejecución cuando se lleve a cabo la destrucción automática de las variables.

Lista de tableros iniciales

La última funcionalidad de la práctica será permitir al usuario jugar una partida aleatoria, o utilizar algún juego ya creado y almacenado en un archivo de texto. **Los juegos almacenados están en su estado inicial, es decir, todas sus celdas están ocultas**.

Cuando empieza la aplicación, se le solicita al usuario el nombre de un archivo de texto. La idea es cargar la información del archivo en una *lista de juegos*, inicialmente vacía, y de la que el usuario podrá seleccionar un juego en su estado inicial. La estructura de estos archivos será la siguiente:

```
numero de juegos (por ejemplo 3)
descripcion del juego 1
descripcion del juego 2
descripcion del juego 3
```

La *descripción* de cada juego es igual que en la version 1 de la práctica. Una vez que el usuario ha dado el nombre del archivo, existen varias posibilidades:

1. El archivo no existe. En este caso obligatoriamente se jugará con un juego generado de forma aleatoria.

2. El archivo existe pero contiene `0` juegos. En este caso de nuevo se jugará con un juego aleatorio.

3. El archivo existe. En tal caso al usuario se le permite elegir si desea seleccionar un juego existente de la lista de juegos, o bien jugar uno aleatorio.

A continuación te mostramos, de forma gráfica, las tres posibilidades:

```
Buscaminas
----------
Nombre del fichero de juegos: noexiste.txt
Error en la apertura del fichero de juegos
Se genera un juego aleatorio....
Numero de filas (>=3) y columnas (>=3) del tablero:

Buscaminas
----------
Nombre del fichero de juegos: vacio.txt
El fichero cargado no tiene juegos... Se crea uno aleatorio...
Numero de filas (>=3) y columnas (>=3) del tablero:

Buscaminas
----------
Nombre del fichero de juegos: partidas.txt
Juego nuevo (opcion 1) o juego existente (opcion 2):
```

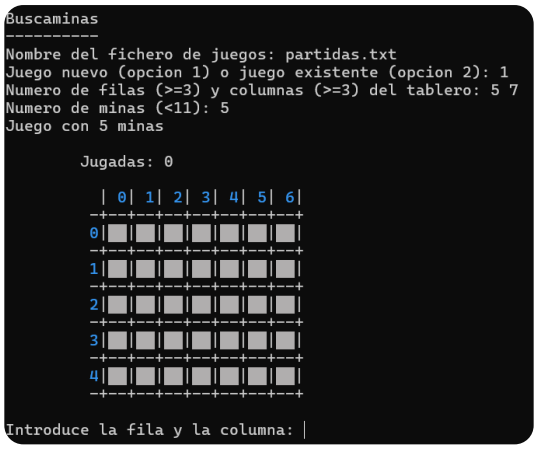
Una vez seleccionado el juego, si el juego es nuevo, debe insertarse en la lista de juegos antes de comenzar a jugar, es decir, debe almacenarse en su estado inicial. El juego transcurre igual que en la versión `1`, excepto que cambia la forma de descubrir las casillas vecinas. Si el juego termina, bien porque se gana o se pierde, entonces ese juego debe eliminarse de la lista de juegos. Si se acaba la partida con `(-1,-1)`, entonces no se hace nada. Siempre que se termina una partida, de la forma que sea, la lista de juegos se almacenará en un archivo de texto, tal y como se explicará más adelante.

Juego aleatorio

Los juegos aleatorios que permitimos deben tener como mínimo `3` filas y `3` columnas, y como máximo `filas*columnas / 3` minas. Deberás añadir a la clase `Juego` la constructora:

```
Juego(int fils, int cols, int numMinas);
```

que crea un juego de dimensión `fils*cols` con `numMinas` colocadas de forma aleatoria en el tablero.



Lista de juegos

Para almacenar la lista de juegos deberás implementar la siguiente clase:

```
class ListaJuegos {
private:
  int cont;
  Juego** lista;
  int capacidad;
public:
  ListaJuegos();
  ~ListaJuegos();
  int insertar(const Juego& juego);
  int dame_longitud() const;
  bool es_vacia() const;
  const Juego& dame_juego(int pos) const;
  void eliminar(int pos);
};
```

Como puedes observar la lista de juegos está implementada como un array dinámico de punteros a `Juego`. Su constructora por defecto crea la lista vacía, y su destructora libera la memoria creada por la clase. **Además la lista estará ordenada de mayor a menor por el orden de dificultad del juego**. Explicaremos con detalle el orden en la descripción de la función `insertar`. Los métodos hacen lo siguiente:

- `dame_longitud()`: devuelve la longitud de la lista.
- `es_vacia()` devuelve `true` si y sólo sí la lista está vacía.
- `dame_juego(pos)`: devuelve una referencia constante al juego que ocupa la posición pos.
- `insertar(juego)`: inserta `juego` de forma ordenada en la lista y devuelve la posición en la que lo ha insertado. Recuerda que el array es dinámico y debes redimensionar en caso de que esté lleno. Además tendrás que crear memoria nueva para añadir el juego, invocando a la constructora por copia de `Juego`.

  La lista está ordenada de mayor a menor por la dificultad de los juegos. La *dificultad* de un juego se calcula como la dimensión de su tablero dividida entre el número

de minas. Lógicamente la lista puede tener repeticiones ya que puede haber varios juegos distintos con la misma dificultad.

- `eliminar(pos)`: elimina de la lista el juego que ocupa la posición `pos`, liberando la memoria ocupada por el juego.

Para implementar esta clase, te será de gran ayuda implementar el método privado:

```
void buscar(const Juego& juego, int& pos) const;
```

que, si existe en la lista un juego con la misma dificultad, devuelve en `pos` la primera posición del juego con esa dificultad. Si no existe ningún juego con esa dificultad, devuelve en `pos` la posición que debería ocupar `juego` en la lista de juegos.

Gestor de juegos

Vamos a crear esta clase para gestionar la interacción entre los juegos y el usuario, liberando a la función `main` de esta tarea. De esta forma el código quedará más claro. La definición de la clase es la siguiente:

```
class GestorJuegos {
private:
  ListaJuegos juegos;
public:
  GestorJuegos();
  bool cargar_juegos();
  void mostrar_lista_juegos() const;
  int numero_juegos() const;
  const Juego& dame_juego(int pos) const;
  int insertar(const Juego& juego);
  void eliminar(int pos);
  bool hay_juegos() const;
  bool guardar_lista_juegos() const;
};
```

La constructora por defecto no hace nada. El método `cargar_juegos()` solicita al usuario el nombre de un archivo del cual cargar los juegos. Si existe el archivo, almacena su contenido en la lista y devuelve `true`. En otro caso devuelve `false`. De la misma forma `guardar_lista_juegos()` solicita al usuario el nombre de un archivo donde almacenar la lista de juegos, y los guarda en ese archivo usando el mismo formato que los archivos de entrada.

El método `mostrar_lista_juegos` muestra la información sobre la dificultad de los juegos almacenados en la lista. Dado que la lista está ordenada de mayor a menor dificultad, este método los mostrará en este orden. Por ejemplo:

```
Buscaminas
----------
Nombre del fichero de juegos: partidas.txt
Juego nuevo (opcion 1) o juego existente (opcion 2): 2
```

```
Mostrando lista de juegos por orden de dificultad....
Juego 0:
   Dimension: 4 x 4
   Minas: 4
Juego 1:
   Dimension: 4 x 4
   Minas: 1
selecciona la partida:
```

Los métodos `insertar(juego)`, `eliminar(pos)`, `numero_juegos()` y `hay_juegos()` utilizan el atributo `juegos` para realizar, de forma natural, su tarea.

Módulos InputOutput y main

El módulo `InputOutput` deberás adaptarlo a la nueva implementación. Es posible que te sobren funciones y que tengas que añadir algunas nuevas. Depende de cómo realices tu implementación.

Con respecto a la función `main` te damos el esqueleto para que completes tú la información:

```
int main() {
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
    srand(time(NULL));
    mostrar_cabecera();
    GestorJuegos GP;

    // añade el código para crear el juego, jugar,
    // y almacenar la lista de juegos
}
```