

PRÁCTICA 2ª: Creación y definición de una clase para el manejo de matrices dinámicas de dos dimensiones.

OBJETIVOS: Introducción al concepto de clase, atributos (datos miembro) y métodos (funciones miembro).

TEMPORIZACIÓN:

Publicación del enunciado: Semana del 9 de septiembre.

Entrega: Semana del 30 de septiembre junto con la práctica 1.

Límite de entrega (con penalización): Semana del 7 de octubre.

BIBLIOGRAFÍA

Programación orientada a objetos con C++

Autor: Fco. Javier Ceballos

Editorial: RA-MA.

Esta práctica deberá utilizar los mecanismos de entrada/salida y de asignación dinámica de memoria propios de C++ y **no los de C**.

Se debe implementar la clase especificada por:

```
class CMatFloat
{
    // Datos privados de la clase

    private:

        float **m_ppDatosF;    // Apunta a los datos de matriz
        int m_nFilas;          // Número de filas
        int m_nColumnas;       // Número de columnas

    // Métodos (funciones miembro) de la clase

    public:

        void Iniciar();
        // Será invocada cada vez que se defina un objeto
        // Pone m_ppDatosF a NULL y m_nFilas y m_nColumnas a 0.

        void CrearMatriz2D(int nFilas, int nColumnas);
        // Asigna memoria para una matriz dinámica cuyas
        // dimensiones vienen dadas por los parámetros de tipo
        // entero que se le pasan y verifica que la asignación fue
        // correcta (en ningún caso deben quedar lagunas de memoria).
        // Pone la matriz a ceros. Asigna a los datos miembro
        // m_nFilas y m_nColumnas los valores adecuados.

        void CrearMatriz1D(int nElementos);
        // Método análogo al anterior pero para una dimensión.
        // Será implementado en función de CrearMatriz2D.
```

```
void Introducir();
// Establece los elementos de la matriz con los valores
// que se introducen por teclado. Valida los datos introducidos
// utilizando la funcionalidad proporcionada por utils.cpp.

void Mostrar();
// Vuelca en la pantalla los datos contenidos en la matriz.
// Mostrar una fila debajo de otra, si procede.

void Destruir();
// Libera la memoria ocupada por los datos y llama a Iniciar.

bool Existe();
// Devuelve true si m_ppDatosF es distinto de NULL
// (la matriz existe); en otro caso, devuelve false.
};
```

Recuerde: utilice new/delete, cin/cout,... en lugar de malloc/free, scanf/printf... Cuando utilice el operador new verifique siempre si la asignación de memoria tuvo éxito; en caso contrario, envíe un mensaje y finalice el programa liberando la memoria que se hubiera asignado hasta entonces.

En prácticas posteriores aprenderá que las tareas realizadas por los métodos Iniciar y Destruir son tareas propias de los constructores y destructores de las clases, por lo que no será necesario implementarlos. Para probarlo, comente (//) las líneas donde se invoca a esos métodos. Coloque la siguiente sentencia a continuación de la que llamaba a Iniciar:

```
system("pause");
```

Añada el constructor y destructor de la clase para que invoquen a esas funciones y verifique cómo son llamadas automáticamente cuando se crea/destruye un objeto:

```
CMatFloat::CMatFloat()
{
    cout << "Se llama a Iniciar\n";
    Iniciar();
}

CMatFloat::~CMatFloat()
{
    cout << "Se llama a Destruir\n";
    if (m_ppfDatos != NULL) Destruir();
}
```

La aplicación deberá mostrar el siguiente menú:

1. Construir matriz 1D
2. Construir matriz 2D
3. Introducir matriz
4. Mostrar matriz
5. Destruir matriz
6. Terminar

Como las operaciones de entrada de datos y su verificación son comunes a todas las prácticas, vamos a crear los ficheros `utils.h` y `utils.cpp` que incluyan la funcionalidad necesaria para poder realizar las operaciones mencionadas. Por ejemplo, `int LeerInt()`, `float LeerFloat()`, `int CrearMenu(char *opciones_menu[], int num_opciones)`, etc. Las funciones `Leer...` devuelven el dato leído del teclado y `CrearMenu` el entero correspondiente a la opción seleccionada del menú (vea en la bibliografía especificada el apartado “Ejercicios resueltos” del capítulo “Excepciones” a modo orientativo, pero no utilice `cin.exceptions` ni `template`). Por ejemplo:

```
int LeerInt()
{
    int error;
    int num;
    do
    {
        cin >> num;
        error = cin.rdstate() & ios::failbit;

        /* ESCRIBA EL CÓDIGO QUE FALTA */

    }
    while(error);
    cin.ignore(numeric_limits<int>::max(), '\n'); // eliminar '\n'
    return num;
}
```

Esta funcionalidad podrá ser realizada utilizando funciones externas o funciones miembro de una clase declaradas **static**.

La aplicación estará compuesta, al menos, por los archivos `CMatFloat.h` y `CMatFloat.cpp` que contendrán la declaración y definición, respectivamente, de la clase `CMatFloat`, por los archivos `utils.h` y `utils.cpp` que contendrán la declaración y definición de, al menos, las funciones `LeerInt`, `LeerFloat` y `CrearMenu`, por el archivo `práctica2.cpp` que contendrá la definición de la función `main` y por los archivos `MemoryManager` indicados en la práctica 1.

¿Podrían los métodos `CrearMatriz2D` y `CrearMatriz1D` llamarse simplemente `CrearMatriz`?

REALIZAR otra versión del programa en la que `m_ppDatosF` sea de tipo `vector<T>` en lugar de `float**`. No implemente los atributos/métodos de `CMatFloat` anteriormente descritos que no sean necesarios al utilizar el tipo `vector<T>` así como cualquier otra funcionalidad no necesaria. Por ejemplo, el método `size()` de `vector<T>` le permitirá conocer las filas y columnas de la matriz, por lo que no es necesario almacenar estos valores. Para acceder a los elementos de la matriz puede utilizar la indexación (por ejemplo, `m_ppDatosF[f][c]`) o iteradores, lo que le resulte más sencillo.

Utilice el juego de pruebas expuesto en el sitio desde donde descargó la práctica para probar el funcionamiento de la misma.