RELACIÓN DE PROBLEMAS II. Memoria dinámica

Problemas sobre *arrays* dinámicos

1. Deseamos guardar un número indeterminado de valores `int`. Resuelva el problema almacenando los datos en un *array dinámico* que vaya creciendo conforme necesite espacio para almacenar un nuevo valor.

   Escribir una función que redimensione el *array* dinámico, ampliándolo, cuando no haya espacio para almacenar un nuevo valor:

   a) *en una casilla*

   b) en *bloques de tamaño* `TAM_BLOQUE`

   c) *duplicando* su tamaño

   La funcion responderá al siguiente prototipo:

   ```
   void  Redimensiona (int * & p, TipoRedimension tipo, int & cap);
   ```

   donde:

   - el primer argumento indica la dirección de memoria del primer elemento del *array* que se va a redimensionar.
   - el segundo argumento indica el tipo de redimensión que se va a aplicar. Proponemos usar un tipo enumerado:

     ```
     enum TipoRedimension {DeUnoEnUno,EnBloques,Duplicando};
     ```

   - el tercer argumento es una **referencia** a la variable que indica la capacidad (número de casillas reservadas) del *array* dinámico. Recuerde que esta operación *modifica* la capacidad del *array*.

   Para la resolución de este ejercicio proponemos que en la función `main`:

   - se reserve incialmente `TAM` casillas (la capacidad será `TAM`),
   - se lea un número indeterminado de valores (terminando cuando se introduzca `FIN`),
   - se añaden los valores al *array* conforme se van leyendo. En el momento de añadir un valor al *array*, si se detecta que no se puede añadir porque el *array* está completo, entonces se llama a la función `Redimensiona` para poder disponer de más casillas.

   Escribir la función `main` para que admita argumentos desde la línea de órdenes de manera que el programa pueda ejecutarse:

   a) Sin argumentos. En este caso, el tipo de redimensión será *de uno en uno*.

   b) Con un argumento. El valor admitido será 1, 2 ó 3 de manera que 1 indica *de uno en uno*, 2 indica *en bloques* y 3 indica *duplicando*.

2. Se van a gestionar las calificaciones de una clase formada por un número indeterminado de alumnos. Se pretende calcular la nota media final de cada alumno en base a un número indeterminado de calificaciones parciales con diferente peso y ordenar los registros de alumnos de acuerdo a la nota media. El número de pesos/notas se calcula durante la lectura de los datos.

   Este ejercicio es un ampliación del ejercicio 19 de la Relación de Problemas I.

   El programa lee, en primer lugar, el número de pesos que se asignan a las calificaciones parciales, y sus valores (en tantos por cien) y comprueba que sumen 100.0 (en el caso de que no fuera así, se aborta la ejecución del programa.

   A continuación lee, para cada alumno, DNI[4], apellidos, nombre y calificaciones (cada uno de los datos por separado). La lectura finaliza cuando se introduce el carácter `*` en la lectura de los apellidos de un alumno.

   Una vez almacenados todos los datos leídos, se calcula la nota media ponderada para cada alumno y se muestra un lista ordenado de acuerdo a la nota media.

   Novedades de este ejercicio respecto al ejercicio 19 de la Relación de Problemas I:

   1. Datos de los alumnos.

      Usen un *array dinámico* de datos `RegAlumno` para guardar los datos de los alumnos. El tipo `RegAlumno` se modifica para este ejercicio de manera que se va a reservar la memoria estrictamente necesaria, tanto para el el número de registros que se almacenan en el `array` dinámico como para almacenar el DNI, el apellido y el nombre, y las notas de cada alumno:

      ```
      struct RegAlumno {
          char   * DNI;
          char   * apellido_nombre;
          double * notas;
      };
      ```

      En definitiva, los punteros `DNI` y `apellido_nombre` referencian a *arrays* dinámicos de datos `char`[5] y el puntero `notas` referencia a un *array* dinámico de datos `double`.

   2. Calificaciones y pesos de calificaciones.

      Los pesos que se emplean para ponderar las calificaciones se almacenarán en un *array* dinámico, independiente, de datos `double`.

      En este problema **no** se limita el número de calificaciones (que coincide con el número de pesos que se asignan a las calificaciones parciales para calcular la nota media).

      Este valor será el primer dato que lea el programa (usar una variable `int`*).*

   Recomendación: Leer los datos usando la redirección de la entrada. Usad para ello un fichero de texto como el disponible en la página de la asignatura.

[4] Este campo es nuevo

[5] Reserve una casilla adicional para almacenar el carácter `'\0'` y poder procesar el *array* dinámico de caracteres como una cadena clásica

3. En el seminario titulado *Modularización del software en C++ (2ª parte). El proyecto* `Poligono Regular` *(ampliación)* presentamos la clase `PoligonoRegular`. Las propiedades de la clase (campos privados de los objetos) se declararon:

```
Punto2D centro;  // Centro de la circunferencia circunscrita
                 // que encierra al polígono.

int     num_lados;  // Num. de lados del polígono
double  longitud;   // Longitud de cada lado

// AMPLIACIÓN de la versión empleada para el examen:
// Los valores de los vértices (las coordenadas)
// también se guardan en el objeto.
// Se emplea para este fín una secuencia de datos "Punto2D".
//
// Es importante tener en cuenta que existe un número
// máximo de vértices, que está determinado por el número
// de casillas disponibles para los datos de la secuencia.
// Puede consultarse con el método "Capacidad()"

SecuenciaPuntos2D vertices;
```

Se trata de modificar la declaración de la clase `PoligonoRegular` para salvar la restricción que imponen las clases`Secuencia...` de tener que especificar a priori la capacidad del *array* que sirve de soporte de almacenamiento a la secuencia.

Emplee para ello un *array* dinámico de datos `Punto2D`:

```
Punto2D centro;  // Centro de la circunferencia circunscrita
                 // que encierra al polígono.
......

Punto2D * vertices; // Array dinámico de datos Punto2D
```

y reescriba adecuadamente el proyecto.

Recomendaciones:

1. El número de vértices del polígono se establece en los constructores, y éstos llaman al método privado `SetVertices` para establecer el valor de los vértices. Es en este método donde tendrá que reservar la memoria necesaria.
2. Deberá modificar todos los métodos que acceden (consulta o modificación) a los valores de los vértices, y antes lo hacían a través de la clase `SecuenciaPuntos2D`.
3. Reflexione acerca de lo que ocurre cuando realiza una asignación entre dos objetos de la clase `PoligonoRegular`. Consulte en los apuntes de clase la diferencia entre *copia profunda* y *copia superficial*.

Problemas sobre matrices dinámicas

4. Supongamos que para definir matrices bidimensionales dinámicas de datos de tipo `TipoBase` usamos una estructura como la que aparece en la figura 22 (tipo `Matriz2D_1`, filas independientes) en la que ilustramos cómo se almacena en memoria una matriz dinámica de 10 filas y 15 columnas.

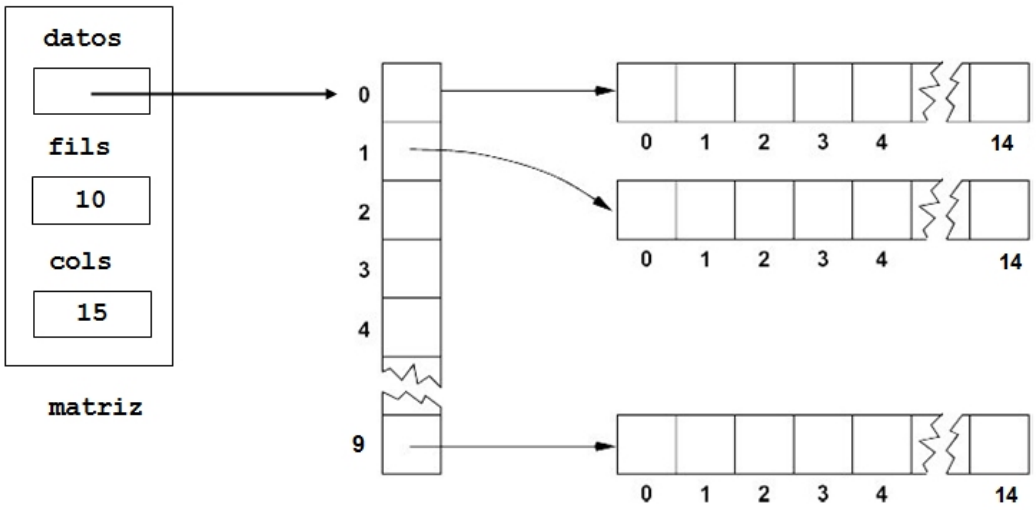

Figura 22: Tipo `Matriz2D_1`: datos guardados en filas independientes

a) Construir una función que cree (reserve memoria para) una matriz de este tipo, recibiendo el número de filas y columnas. El contenido de la matriz queda *indefinido*.

b) Construir una función que libere la memoria ocupada por una matriz de este tipo. La función deja la matriz *vacía*.

c) Construir una función que cree una matriz (como en el apartado 4*a*) pero además lea del teclado los valores y los copie en la matriz. La matriz se rellena completamente.

d) Construir una función que cree una matriz (como en el apartado 4*a*) y rellene todas las casilla con valores aleatorios.

e) Construir una función que muestre los valores guardados en la matriz.

f) Construir una función que reciba una matriz de ese tipo, y cree y devuelva una copia (copia *profunda*). La copia es una *nueva* matriz.

g) Construir una función que extraiga una submatriz. Como argumento de la función se introduce desde qué fila y columna y hasta qué fila y columna se debe realizar la copia de la matriz original. La submatriz devuelta es una *nueva* matriz.

h) Construir una función que elimine una fila de una matriz. Obviamente, no se permiten "huecos" (filas vacías). La función devuelve una *nueva* matriz.

i) Construir una función como el anterior, pero que en vez de eliminar una fila, elimine una columna. La función devuelve una *nueva* matriz.

j) Construir una función que devuelva (*nueva* matriz) la traspuesta de una matriz.

k) Construir una función que reciba una matriz y devuelva una *nueva* matriz con las filas "inviertidas": la primera fila de la nueva matriz será la última fila de la primera, la segunda será la penúltima, y así sucesivamente.

Modularice la solución en dos ficheros: `Matriz2D_1.h` y `Matriz2D_1.cpp` y escriba un fichero con una función `main` que ilustre el uso de las funciones.

5. Supongamos que ahora decidimos utilizar una forma diferente para representar las matrices bidimensionales dinámicas a la que se propone en el ejercicio 4. Usaremos una estructura semejante a la que aparece en la figura 23 (tipo `Matriz2D_2`, todas las casillas consecutivas formando una única fila) en la que ilustramos cómo se almacena en memoria una matriz dinámica de 10 filas y 15 columnas.

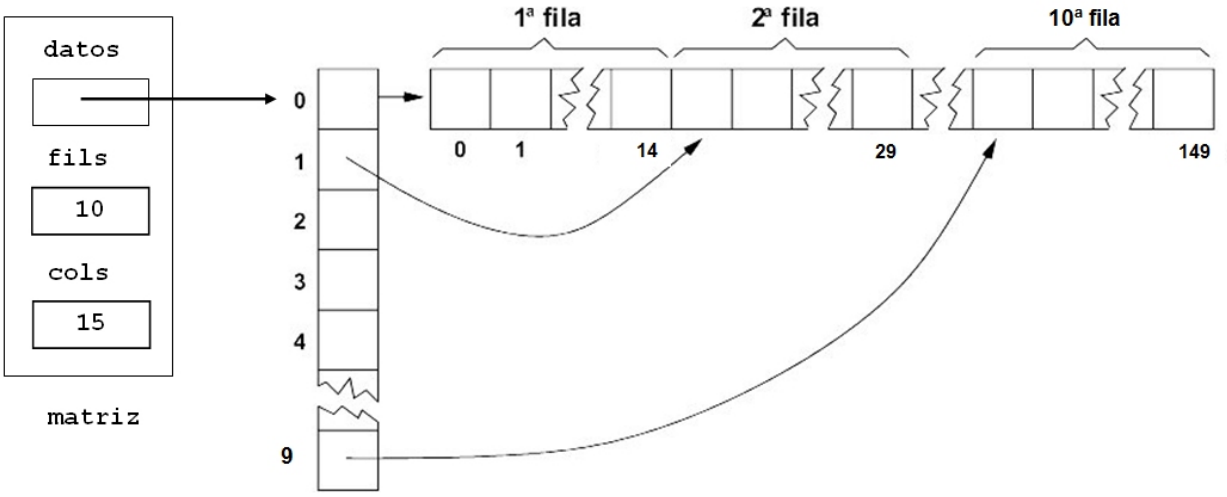

Figura 23: Tipo `Matriz2D_2`: datos guardados en una sola fila

Reescribir todos los módulos propuestos en el ejercicio 4.

Modularice la solución en dos ficheros: `Matriz2D_2.h` y `Matriz2D_2.cpp` y escriba un fichero con una función `main` que ilustre el uso de las funciones.

Nota: La función `main` debería ser, básicamente, la misma que empleó en el ejercicio 4.

6. Una vez implementados los módulos que gestionan matrices dinámicas bidimensionales (ejercicios 4 y 5),

   a) Construir una función que, dada una matriz `Matriz2D_1`, realice una copia de la misma en una matriz `Matriz2D_2` y la devuelva.

   b) Desarrollar una función que realice el paso inverso, convertir de `Matriz2D_2` a `Matriz2D_1` y la devuelva.

   Modularice la solución en dos ficheros: `Conversiones-Matriz2D.h` y y `Conversiones-Matriz2D.cpp` y escriba un fichero con una función `main` que ilustre el uso de las funciones.

Problemas sobre listas

Los siguientes ejercicios gestionan listas enlazadas. Hay una serie de tareas comunes (crear listas, liberar la memoria ocupada, contar el número de nodos, etc.) que se repiten en muchos de los ejercicios, o que podrían ser útiles en otros programas que gestionaran listas.

Con objeto de evitar duplicar código, construyan la biblioteca `libLista.a` que ofrezca a través de `Lista.h` los *tipos de datos* y las *funciones* que considere adecuadas para la gestión de listas enlazadas.

Esta biblioteca se enlazará con el fichero objeto que contiene la función `main` que resuelve cada uno de los ejercicios indicados para generar el correspondiente ejecutable.

La biblioteca se basa en los módulos **Lista** y **Random**:

- **Lista**: Ficheros `Lista.h` (declaración de los tipos `PNodo` y `Lista`, y los prototipos de las funciones) y `Lista.cpp` (implementación de las funciones).

  Los tipos `PNodo` y `Lista` son *alias* de tipos *puntero a* `Nodo`:

```
// Tipo de los elementos de la lista
// Bastará cambiar el tipo asociado al alias TipoBase
// y recompilar para poder gestionar otro tipo de lista.

typedef double TipoBase;

// Cada nodo de la lista es de tipo "Nodo"

struct Nodo {
   TipoBase valor;
   Nodo *sig;
};

typedef Nodo * PNodo;  // Para los punteros a nodos
typedef Nodo * Lista   // Para la lista
```

- **Random**: Ficheros `Random.h` y `Random.cpp`, que contienen la declaracción y definición, respectivamente, de la clase `MyRandom`.

En definitiva, el fichero de biblioteca `libLista.a` *se construirá con* `Lista.o` *y* `Random.o`.

7. Escriba un programa para que lea una secuencia con un número indefinido de valores `double` hasta que se introduzca un valor negativo. Estos valores (excepto el último, el negativo) los almacenará en una estructura de celdas enlazadas (una *lista*) y después mostará los valores almacenados.

   Escribir un programa para solucionar este problema.

   > **Nota:** Deberá implementar una función para liberar la memoria ocupada. Esta función se empleará en **todos** los programas que gestionen listas.
   >
   > **Nota:** Es aconsejable implementar una función que compruebe si la lista está vacía.

   Ampliar el ejercicio rellenando una lista con valores aleatorios. Usad para ello una función específica.

   Funciones a implementar (sugerencia):

```
void LeeLista (Lista & l);
void PintaLista (const Lista l);
void LiberaLista (Lista & l);
bool ListaVacia (const Lista l);
void RellenaListaAleatoriamente (Lista & l, int num_datos,
                                 int min, int max);
```

   La última función rellena la lista `l` con `num_datos` valores aleatorios entre `min` y `max`.

8. Utilizando como base la solución al ejercicio 7 escriba un programa que una vez leídos los datos y guardados en la lista (o generados aleatoriamente) realice unos cálculos sobre los datos almacenados en la lista:

   a) el número de celdas enlazadas.

   b) la media de los datos almacenados.

   c) la varianza de los datos almacenados.

   Funciones a implementar (sugerencia):

```
int    CuentaElementos (const Lista l);
double Media (const Lista l);
double Varianza (const Lista l);
```

9. Escribir un programa que lea una secuencia de valores, los almacene en una *lista* y determine si la secuencia está ordenada.

   Funciones a implementar (sugerencia):

   ```
   bool   EstaOrdenada (const Lista l);
   ```

10. Escribir un programa que lea una secuencia de valores (o los genere aleatoriamente), los almacene en una *lista* y los ordene.

    La ordenación se debe efectuar sobre la propia lista, sin emplear ninguna estructura de datos auxiliar (array dinámico, otra lista, etc.) simplemente cambiando la posición de los nodos.

    Utilice, por ejemplo, el método de *ordenación por selección*.

    Funciones a implementar (sugerencia):

    ```
    void   OrdenaSeleccionLista (Lista &l);
    ```

11. Considere una secuencia **ordenada** de datos almacenada en una *lista* (dispone de funciones para leer o generar una lista con valores aleatorios y ordenarla).

    a) Implemente una función para insertar un nuevo dato en su posición correcta. En el caso que la lista ya tuviera ese valor, se insertará el nuevo delante de la primera aparición de éste.

    b) Implemente una función para, dado un dato, eliminar la celda que lo contiene. En el caso que la lista tuviera ese valor repetido, se eliminará la primera aparición de éste.

    Funciones a implementar (sugerencia):

    ```
    void   InsertaOrdenadamente (Lista &l, TipoBase v);
    void   EliminaValor (Lista &l, TipoBase v);
    ```

12. Considere dos secuencias de datos **ordenadas** almacenadas en sendas *listas*. Implemente una función para *mezclar ordenadamente* las dos secuencias en una nueva, de forma que las dos listas originales se queden *vacías* tras realizar la mezcla y la lista resultante contenga todos los datos.

    Observe se trata de una variante del algoritmo *mergesort*. Ahora se exige la modificación de las secuencias originales: en esta versión los datos se “mueven” hacia la lista resultante en lugar de copiarlos.

    Nota: No es preciso (ni se permite) realizar ninguna operación de reserva ni liberación de memoria.

    Funciones a implementar (sugerencia):

    ```
    void   MezclaListas (Lista &l, Lista &l1, Lista &l2);
    ```