Metodología de la Programación II

Gestión Dinámica de Memoria (Ampliación)

1.1 Introducción

La utilización de variables estáticas o automáticas para almacenar información cuyo tamaño no es conocido a priori (solo se conoce exactamente en tiempo de ejecución) resta generalidad al programa.

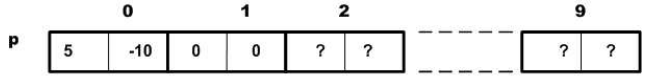
Ejemplo: Polígono (1)

Se desea desarrollar una estructura de datos que permita representar de forma general diversas figuras poligonales. Cada figura poligonal puede representarse como un conjunto de puntos en el plano unidos por segmentos de rectas entre cada dos puntos adyacentes.

VERSIÓN 1: VECTOR ESTÁTICO

La estructura de datos que soporta el tipo de datos `TPoligono` es muy simple: un vector estático de registros de tipo TPunto.

```
struct TPunto2D {
   int x;
   int y;
};

const int MAX = 10;
typedef TPunto2D TPoligono[MAX];

......

TPoligono p;
```

`defs1.h`

```
#ifndef DEFS1
#define DEFS1

   const int MAX = 10;

   struct TPunto2D {
      int x;
      int y;
   };
   typedef TPunto2D TPoligono[MAX];

#endif
```

ver1.cpp

```
/*************************************/
// Version 1: VECTOR ESTATICO(1)
/*************************************/
using namespace std;

#include <iostream>
#include <iomanip>
#include <cstdlib>

#include "defs1.h"


void RellenaPoligono (TPoligono pol, int coordenadas[][2], int NumLados);
void PintaPoligono (const char * msg, TPoligono pol, int NumLados);
```



```
int main (void)
{
   int  cuadrado[][2] = {{0,0},{5,0},{5,5},{0,5}};
   int triangulo[][2] = {{2,0},{4,0},{3,1}};
   TPoligono Poligono1, Poligono2, Poligono3;

   RellenaPoligono (Poligono1, cuadrado, 4);
   PintaPoligono ("Cuadrado", Poligono1, 4);

   RellenaPoligono (Poligono2, triangulo, 3);
   PintaPoligono ("Triangulo 1", Poligono2, 3);

   RellenaPoligono (Poligono3, cuadrado, 3);
   PintaPoligono ("Triangulo 2", Poligono3, 3);

   return (0);
}
```



```
void RellenaPoligono (TPoligono pol, int coordenadas[][2], int NumLados)
{
   TPunto2D UnPunto;

   if (NumLados <= MAX) {
      for (int lado=0; lado<NumLados; lado++) {
         UnPunto.x = coordenadas[lado][0];
         UnPunto.y = coordenadas[lado][1];
         pol[lado] = UnPunto;
      }
   }
   else {
      cerr << "\n   Error: no se pudo inicializar ";
      cerr << "el poligono (demasiados puntos)\n\n";
      exit (1);
   }
}
```



```
void PintaPoligono (const char* msg, TPoligono pol, int NumLados)
{
  cout << "\n" << msg << ": \n";
  for (int lado=0; lado<NumLados; lado++) {
     cout << "   (" << setw(3)<< pol[lado].x;
     cout << "," << setw(3) << pol[lado].y << ")\n";
  }
  cout << "   -->" << setw(3) << NumLados;
  cout << " vertices.\n";
}
```

- El número de casillas es fijo (`MAX`) y permanece inalterado durante la ejecución del programa, por lo que los polígonos así construidos no pueden tener más de `MAX` vértices.

  Se verá más adelante

- No se guarda el número de vértices de un polígono: las funciones `RellenaPoligono()` y `PintaPoligono()` requieren este valor como parámetro.

Mejoras que pueden realizarse:

1. Puede reservarse una casilla (tipo `TPunto2D`) para guardar el número de vértices del polígono.

   En este problema es factible porque los campos del registro son numéricos (no es una solución recomendable).

2. Puede iniciarse un polígono antes de usarse con valores "imposibles".

   En este caso, dado que se guardan coordenadas y éstas, potencialmente, no están restringidas, hay que realizar alguna suposición para limitar el rango de éstas. Por ejemplo: las coordenadas son siempre positivas, las coordenadas están en un rango, etc.

Con la segunda estrategia:

- Se requiere una función de inicialización antes de usar un polígono.
- Debe controlarse el número de casillas ocupadas.

En cualquier caso: *el tamaño del polígono (número de vértices) está limitado por la declaración -en tiempo de compilación- y no puede cambiarse.*

VERSIÓN 2: VECTOR ESTÁTICO (2)

Implementa las mejoras propuestas para la versión 1:

- Valor "imposible":

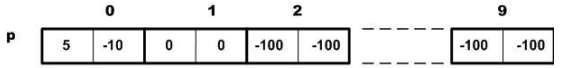
```
const int VACIO = -100;
```

- Función de iniciación:

```
void IniciaPoligono (TPoligono p);
```

- Control de casillas ocupadas: Ver funciones `AniadePunto()` y `PintaPoligono()`

En cualquier caso: *el tamaño del polígono sigue limitado: la estructura de datos (representación) no cambia, solo el modo de acceder a ella.*



`defs2.h`

```
#ifndef DEFS2
#define DEFS2
   const int MAX = 10;
   const int VACIO = -100;

   struct TPunto2D {
      int x;
      int y;
   };
   typedef TPunto2D TPoligono[MAX];
#endif
```

ver2.cpp

```
/*****************************************/
// Version 2: VECTOR ESTATICO(2)
/*****************************************/
using namespace std;

#include <iostream>
#include <iomanip>
#include <cstdlib>

#include "defs2.h"

void RellenaPoligono (TPoligono pol, int coordenadas[][2], int NumLados);
void PintaPoligono (const char * msg, TPoligono pol);

void IniciaPoligono (TPoligono p);
void AniadePunto(TPoligonopol, TPunto2D punto);
```



```
int main (void) {
   int  cuadrado[][2] = {{0,0},{5,0},{5,5},{0,5}};
   int triangulo[][2] = {{2,0},{4,0},{3,1}};
   TPoligono Poligono1, Poligono2, Poligono3;

   IniciaPoligono(Poligono1);
   IniciaPoligono(Poligono2);
   IniciaPoligono(Poligono3);

   RellenaPoligono (Poligono1, cuadrado, 4);
   PintaPoligono ("Cuadrado", Poligono1);

   RellenaPoligono (Poligono2, triangulo, 3);
   PintaPoligono ("Triangulo 1", Poligono2);

   RellenaPoligono (Poligono3, cuadrado, 3);
   PintaPoligono ("Triangulo 2", Poligono3);
```



```
   // Aniadir un punto a un poligono es sencillo:
   TPunto2D PuntoNuevo = {0, 10};
   AniadePunto (Poligono3, PuntoNuevo);
   PintaPoligono ("Poligono 3 (actualizado)", Poligono3);

   // Aniadimos dos nuevos puntos
   PuntoNuevo.x = -5;
   PuntoNuevo.y =  5;
   AniadePunto (Poligono3, PuntoNuevo);

   PuntoNuevo.x = -5;
   PuntoNuevo.y =  0;
   AniadePunto (Poligono3, PuntoNuevo);

   PintaPoligono ("Poligono 3 (actualizado 2)", Poligono3);
   return (0);
}
```



```
/*****************************************************/

void IniciaPoligono (TPoligono pol)
{

   for (int i=0; i<MAX; i++)

      pol[i].x=pol[i].y=VACIO;
}

/*****************************************************/

void RellenaPoligono (TPoligono pol, int coordenadas[][2], int NumLados)
{

   TPunto2D UnPunto;
```

```
   if (NumLados <= MAX) {
      for (int lado=0; lado<NumLados; lado++) {
         UnPunto.x = coordenadas[lado][0];
         UnPunto.y = coordenadas[lado][1];
         pol[lado] = UnPunto;
      }
   }
   else {
      cerr << "\n   Error: no se pudo inicializar ";
      cerr << "el poligono (demasiados puntos)\n\n";
      exit (1);
   }
}
```



```
/*********************************************************/

void PintaPoligono (const char * msg, TPoligono pol);
{
   int lado;
   cout << "\n" << msg << ": \n";
   for (lado=0; pol[lado].x != VACIO; lado++) {
      cout << "   (" << setw(3)<< pol[lado].x;
      cout << "," << setw(3) << pol[lado].y
           << ")\n";
   }
   cout << "   -->" << setw(3) << lado;
   cout << " vertices.\n";
}

/*********************************************************/
```

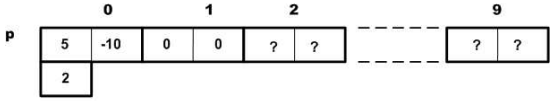


```
void AniadePunto (TPoligono pol, TPunto2D punto)
{
   int nv;
   for (nv=0; pol[nv].x != VACIO; nv++);

   if (nv < MAX) {
      pol[nv].x = punto.x;
      pol[nv].y = punto.y;
   }
   else {
      cerr << "\n   Error: no se pudo aniadir ";
      cerr << "el punto (demasiados puntos)\n\n";
      exit (1);
   }
}
```


VERSIÓN 3: STRUCT Y VECTOR ESTÁTICO

Vértices en un vector estático: la restricción de limitación persiste.

```
struct TPunto2D {
   int x;
   int y;
};

const int MAX = 10;
typedef TPunto2D TVectorPuntos[MAX];

struct TPoligono {
   int NumVertices;
   TVectorPuntos Vertices;
};
......
TPoligono p;
```



- La función de inicialización puede cambiarse (puede devolver un `struct`):

```
TPoligono IniciaPoligono (void);
```

- El número de casillas ocupadas es fácil de controlar: campo `NumVertices`.

En las funciones `RellenaPoligono()` y `AniadePunto()` se modifica la lista de parámetros: el `TPoligono` (es un `struct`) se pasa por referencia. Antes (versiones 1 y 2) era un vector por lo que era pasado por valor.

Ejercicio: `void QuitaPunto (TPoligono & pol, int VertAQuitar);`

`defs3.h`

```
#ifndef DEFS3
#define DEFS3
   const int MAX = 10;
   struct TPunto2D {
      int x;
      int y;
   };

   typedef TPunto2D TVectorPuntos[MAX];
   struct TPoligono {
      int NumVertices;
      TVectorPuntos Vertices;
   };
#endif
```


`ver3.cpp`

```
/*****************************************/
// Version 3: STRUCT Y VECTOR ESTATICO
/*****************************************/
using namespace std;

#include <iostream>
#include <iomanip>
#include <cstdlib>

#include "defs3.h"

void RellenaPoligono (TPoligono & pol, int coordenadas[][2], int NumLados);
void PintaPoligono (const char * msg, TPoligono pol);

void AniadePunto(TPoligono & pol, TPunto2D punto);
TPoligono IniciaPoligono (void);
```



```
int main (void)
{
   int  cuadrado[][2] = {{0,0},{5,0},{5,5},{0,5}};
   int triangulo[][2] = {{2,0},{4,0},{3,1}};

   TPoligono Poligono1, Poligono2, Poligono3;

   RellenaPoligono (Poligono1, cuadrado, 4);
   PintaPoligono ("Poligono 1", Poligono1);

   RellenaPoligono (Poligono2, triangulo, 3);
   PintaPoligono ("Poligono 2", Poligono2);

   RellenaPoligono (Poligono3, cuadrado, 3);
   PintaPoligono ("Poligono 3", Poligono3);
```

```
   // Añadir un punto a un poligono es sencillo:

   TPunto2D PuntoNuevo = {0, 10};
   AniadePunto (Poligono3, PuntoNuevo);
   PintaPoligono ("Poligono 3 (actualizado)", Poligono3);

   // Un poligono vacio

   TPoligono p = IniciaPoligono();
   PintaPoligono ("Poligono p", p);

   return (0);
}
```



```
TPoligono IniciaPoligono (void)
{
   TPoligono p;
   p.NumVertices = 0;
   return (p);
}

/*************************************************************************/

void RellenaPoligono (TPoligono & pol, int coordenadas[][2], int NumLados)
{
   TPunto2D UnPunto;

   if (NumLados <= MAX) {

      pol.NumVertices = NumLados;
```



```
      for (int lado=0; lado<NumLados; lado++) {
         UnPunto.x = coordenadas[lado][0];
         UnPunto.y = coordenadas[lado][1];
         pol.Vertices[lado] = UnPunto;
      }
   }
   else {
      cerr << "\n   Error: no se pudo inicializar ";
      cerr << "el poligono (demasiados puntos)\n\n";
      exit (1);
   }
}

/*************************************************************************/
```



```
/*************************************************************************/

void PintaPoligono (const char * msg, TPoligono pol)
 {
   cout << "\n" << msg << ": \n";

   for (int lado=0; lado<pol.NumVertices; lado++)
   {
      cout << "   (" << setw(3)<< pol.Vertices[lado].x;
      cout << "," << setw(3) << pol.Vertices[lado].y
           << ")\n";
   }

   cout << "   -->" << setw(3) << pol.NumVertices;
   cout << " vertices.\n";
}
/*************************************************************************/
```

```
/*************************************************************************/

void AniadePunto (TPoligono & pol, TPunto2D punto)
 {
   if (pol.NumVertices < MAX) {
      pol.Vertices[pol.NumVertices] = punto;
      pol.NumVertices++;
   }
   else {
      cerr << "\n   Error: no se pudo aniadir ";
      cerr << "punto (demasiados puntos)\n\n";
      exit (1);
   }
}

/*************************************************************************/
```


La alternativa consiste en la posibilidad de, **en tiempo de ejecución**: pedir la memoria necesaria para almacenar la información y de liberarla cuando ya no sea necesaria.

Esta memoria se reserva en el Heap y, habitualmente, se habla de **variables dinámicas** para referirse a los bloques de memoria del Heap que se reservan y liberan en tiempo de ejecución.

¡ RECORDAR LA METODOLOGÍA !

1. Reservar memoria.
2. ***Controlar posible fallo de memoria***.
3. Utilizar memoria reservada.
4. Liberar memoria reservada.

1.2 Objetos dinámicos simples

El operador `new (nothrow)`

```
#include <new>
......
```

tipo `* p;`
`p = new (nothrow)` *tipo*`;`

- `new` reserva una zona de memoria en el Heap del tamaño adecuado para almacenar un dato del tipo *tipo* (`sizeof(`*tipo*`)` bytes), devolviendo la dirección de memoria donde empieza la zona reservada.
- Si `new (nothrow)` no puede reservar espacio (p.e. no hay suficiente memoria disponible), devuelve la dirección nula (**0**).

Ejemplo:

```
   #include <new>
   ......

(1)   int *p, q = 10;

(2)   p = new (nothrow) int;
(3)   if (p==0) {   // if (!p)
         cerr << "Error en la reserva\n";
         exit(1);
      }
(4)   *p = q;
```

Notas:

1. Observar que p se declara como un puntero más.
2. Se pide memoria en el Heap para guardar un dato `int`. Si hay espacio para satisfacer la petición, p apuntará al principio de la zona reservada por `new`. Si no nay espacio, p tomará el valor 0.
3. Comprobar **siempre** si se ha tenido éxito.
4. Se trabaja como ya sabemos con el objeto referenciado por p.

El operador `delete`

`delete` *puntero*;

`delete` permite liberar la memoria del Heap que previamente se había reservado y que se encuentra referenciada por un puntero.

Ejemplo:

```
   #include <new>
      ......
(1)   int *p, q = 10;
(2)   p = new (nothrow) int;
(3)   if (p==0) {   // if (!p)
         cerr << "Error en la reserva\n";
         exit(1);
      }
(4)   *p = q;
      ......
(5)   delete p;
```

5. El objeto referenciado por por p deja de ser "operativo" y la memoria que ocupaba está disponible para nuevas peticiones con `new`.

1.3 Objetos dinámicos compuestos

Para el caso de objetos compuestos (p.e. `struct`) la metodología a seguir es la misma.

En el caso de los `struct`, la instrucción `new` reserva la memoria necesaria para almacenar todos y cada uno de los campos de la estructura.

```
struct Persona{
   string nombre;
   string DNI;
};
Persona *yo;
```



```
yo = new (nothrow) Persona;

if (yo == 0) {
   cerr << "Error en la reserva\n";
   exit(1);
}

cin >> (*yo).nombre;  // cin >> yo->nombre;
cin >> (*yo).DNI;     // cin >> yo->DNI;

......

delete yo;
```

1.4 Vectores dinámicos

- Hasta ahora sólo podemos crear un vector conociendo *a priori* el número mínimo de elementos que podrá tener. P.e. `int vector[22];`
- Para reservar la memoria estrictamente necesaria:

  El operador `new (nothrow) []`

  ```
  #include <new>
  ......

  tipo * p;
  p = new (nothrow) tipo[ num ];
  ```


- Reserva una zona de memoria en el Heap para almacenar num datos de tipo *tipo*, devolviendo la dirección de memoria inicial.
- Si no puede, devuelve la dirección nula (**0**).

- La liberación se realiza con `delete []`

  `delete []` *puntero*`;`

  que libera (pone como disponible) la zona de memoria **previamente reservada** por una orden `new []` y que está referencia por un punetro.

- Con la utilización de de esta forma de reserva dinámica podemos crear vectores que tengan justo el tamaño necesario. Podemos, además, crearlo justo en el momento en el que lo necesitamos y destruirlo cuando deje de ser útil.

Ejemplo:

```
#include <new>
#include <iostream>
using namespace std;

int main(void)
{
   int *v, n;
   cout << "Numero de casillas: ";
   cin >> n;

   // Reserva de memoria
   v = new (nothrow) int [n];
```



```
if (!v) {
   cerr << "Error en la reserva\n";
   exit (1);
}

// Procesamiento del vector dinamico:
//    lectura y escritura de su contenido

for (int i= 0; i<n; i++) {
   cout << "Valor en casilla "<<i<< ": ";
   cin >> v[i];
}
cout << endl;
```

```
   for (int i= 0; i<n; i++)
      cout << "En la casilla " << i<< " guardo: "<< v[i]<< endl;

   // Liberar memoria

   delete [] v;

   return (0);
}
```


Ejemplo: Una función que devuelve un vector como copia de otro que recibe como argumento.

```
int * copia_vector1 (int v[], int n)
{
   int * copia = new (nothrow) int [n];
   if (copia) { // Si tiene exito
      for (int i= 0; i<n; i++)
         copia[i] = v[i];
   }
   return (copia);
}
```


El vector devuelto tendrá que ser, cuando deje de ser necesario, liberado con `delete []`:

```
   int * v1_copia;
   ......
   v1_copia = copia_vector1 (v1, 500);
   if (!v1_copia) {
      cerr << "Error de memoria\n";
      exit (1);
   }
   // Usamos v1_copia
   ...
   delete [] v1_copia;
```


Un **error** muy común a la hora de construir una función que copie un vector es el siguiente:

```
int *copia_vector1(int v[], int n)
{
   int copia[100]; // o cualquier otro valor
                   // mayor que n

   for (int i= 0; i<n; i++)
      copia[i] = v[i];

  return (copia);
}
```

Al ser `copia` una variable local no puede ser usada fuera del ámbito de la función en la que está definida.

Ejemplo: ampliación del espacio ocupado por un vector dinámico.

```
void AmpliarVector (int &v[], int nelems,
                    int nadicional)
{
   int * v_ampliado = new (nothrow) int[nelems + nadicional];
   if (!v_ampliado) {
      cerr << "Error de memoria\n";
      exit (1);
   }

   for (int i= 0; i<nelems; i++)
      v_ampliado[i] = v[i];

   delete [] v;
   v = v_ampliado;
}
```


Cuestiones a tener en cuenta:

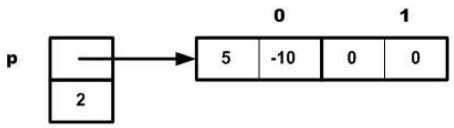
- v se pasa por referencia porque se va a modificar el lugar al que apunta.
- Es necesario liberar v antes de asignarle el mismo valor de v_copia ya que si no perderíamos cualquier referencia a ese espacio de memoria.
- Poco eficiente. Probar con memcpy().

Ejemplo: Polígono (2)

VERSIÓN 4: STRUCT Y VECTOR DINÁMICO

Observar los prototipos de las funciones: idénticos a los de la versión 3. La representación (estructura de datos) es, no obstante, muy diferente.

```
struct TPunto2D {
    int x;
    int y;
};
struct TPoligono {
   int NumVertices;
   TPunto2D * Vertices;
};
......
TPoligono p;
```

La única restricción está ahora en la *cantidad de memoria disponible* para albergar vértices.

Un problema de eficiencia: cada vez que se añade un punto hay que reorganizar **seriamente** la estructura de datos.

Ejercicio: Función que elimina un vértice del poligono.

```
void QuitaPunto (TPoligono & pol, int VertAQuitar);
```

defs4.h

```
#ifndef DEFS4
#define DEFS4

   struct TPunto2D {
      int x;
      int y;
   };
   struct TPoligono {
      int NumVertices;
      TPunto2D * Vertices;
   };
#endif
```


ver4.cpp

```
/*****************************************/
// Version 4: STRUCT Y VECTOR DINAMICO
/*****************************************/

using namespace std;

#include <iostream>
#include <iomanip>
#include <new>
#include <cstdlib>

#include "defs4.h"

/*************************************************************/
```



```
void RellenaPoligono(TPoligono & pol, int coordenadas[][2], int NumLados);
void PintaPoligono (const char * msg, TPoligono pol);

void AniadePunto (TPoligono & pol, TPunto2D punto);
TPoligono IniciaPoligono (void);

/*************************************************************/

int main (void)
{

   int  cuadrado[][2] = {{0,0},{5,0},{5,5},{0,5}};
   int triangulo[][2] = {{2,0},{4,0},{3,1}};

   TPoligono Poligono1 = IniciaPoligono();
   RellenaPoligono (Poligono1, cuadrado, 4);
   PintaPoligono ("Poligono 1", Poligono1);
```



```
   TPoligono Poligono2 = IniciaPoligono();
   RellenaPoligono (Poligono2, triangulo, 3);
   PintaPoligono ("Poligono 2", Poligono2);

   TPoligono Poligono3 = IniciaPoligono();
   RellenaPoligono (Poligono3, cuadrado, 3);
   PintaPoligono ("Poligono 3", Poligono3);

   // Aniadir un punto a un poligono es sencillo:

   TPunto2D PuntoNuevo = {0, 10};

   AniadePunto (Poligono3, PuntoNuevo);
   PintaPoligono ("Poligono 3 (actualizado)", Poligono3);
```

```
   // Aniadimos dos nuevos puntos

   PuntoNuevo.x = -5;
   PuntoNuevo.y =  5;
   AniadePunto (Poligono3, PuntoNuevo);

   PuntoNuevo.x = -5;
   PuntoNuevo.y =  0;
   AniadePunto (Poligono3, PuntoNuevo);
   PintaPoligono ("Poligono 3 (actualizado 2)", Poligono3);

   // Un poligono vacio
   TPoligono p = IniciaPoligono();
   PintaPoligono ("Poligono p", p);

   return (0);
}
```



```
/*************************************************************/

TPoligono IniciaPoligono (void)
{
   TPoligono pol = {0, 0};
   return (pol);
}
/*************************************************************/

void RellenaPoligono (TPoligono & pol, int coordenadas[][2], int NumLados)
{
   pol.NumVertices = NumLados;
   pol.Vertices = new (nothrow) TPunto2D[NumLados];
   if (!pol.Vertices) {
      cerr << "\n   Error: no se pudo iniciar poligono (no hay memoria)\n\n";
      exit(1);
   }
```



```
   else {
      TPunto2D UnPunto;
      for (int lado=0; lado<NumLados; lado++) {
         UnPunto.x = coordenadas[lado][0];
         UnPunto.y = coordenadas[lado][1];
         pol.Vertices[lado] = UnPunto;
      }
   }
}
/*************************************************************/

void PintaPoligono (const char * msg, TPoligono pol) {
   cout << "\n" << msg << ": \n";
   for (int lado=0; lado<pol.NumVertices; lado++) {
      cout << "   (" << setw(3)<< pol.Vertices[lado].x;
      cout << "," << setw(3) << pol.Vertices[lado].y << ")\n";
   }
```



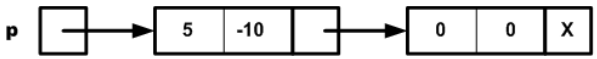
```
   cout << "   -->" << setw(3) << pol.NumVertices;
   cout << " vertices.\n";
}

/*************************************************************/

void AniadePunto (TPoligono & pol, TPunto2D punto)
{
   TPunto2D * aux = new (nothrow) TPunto2D[pol.NumVertices + 1];
   if (!aux) {
      cerr << "\n   Error: no se pudo aniadir ";
      cerr << "punto (no hay memoria)\n\n";
      exit (1);
   }
   else { // Copia en "aux" y aniade el nuevo
      for (int l=0; l<pol.NumVertices; l++)
         aux[l] = pol.Vertices[l];
```

```
      aux[pol.NumVertices] = punto;

      delete [] pol.Vertices; // Libera "pol"

      // Actualiza "pol" copiando lo de "aux"
      pol.NumVertices++;
      pol.Vertices = aux;
   }
}
```


VERSIÓN 5: LISTA ENLAZADA

Observar los prototipos de las funciones: idénticos a los de las versiones 3 y 4. La representación es muy diferente a ambas.

```
  struct TPunto2D {
      int x;
      int y;
   };
   struct TNodo {
      TPunto2D Punto;
      TNodo * sig;
   };
   typedef TNodo* TPoligono;
   ......
   TPoligono p;
```

La única restricción sigue siendo la *cantidad de memoria disponible* para albergar vértices.

En esta implementación hay un "problema" de eficiencia cada vez que se añade un punto porque la adición se realiza al final de la lista: hay soluciones muy variadas que solventan este problema (asignatura *Estructuras de Datos*).

defs5.h

```
#ifndef DEFS5
#define DEFS5
   struct TPunto2D {
      int x;
      int y;
   };
   struct TNodo {
      TPunto2D Punto;
      TNodo * sig;
   };
   typedef TNodo* TPoligono;
#endif
```

ver5.cpp

```
/*****************************************/
// Version 5: LISTA ENLAZADA
/*****************************************/

using namespace std;

#include <iostream>
#include <iomanip>
#include <new>
#include <cstdlib>

#include "defs5.h"

/*****************************************************************/
```



```
TPoligono IniciaPoligono (void);

void RellenaPoligono (TPoligono & pol, int coordenadas[][2], int NumLados);
void PintaPoligono (const char * msg, TPoligono pol);
void AniadePunto (TPoligono & pol, TPunto2D punto);
TNodo * CreaNodo (TPunto2D p);

/*****************************************************************/

int main (void)
{
   int  cuadrado[][2] = {{0,0},{5,0},{5,5},{0,5}};
   int triangulo[][2] = {{2,0},{4,0},{3,1}};

   TPoligono Poligono1 = IniciaPoligono();
   RellenaPoligono (Poligono1, cuadrado, 4);
   PintaPoligono ("Poligono 1", Poligono1);
```



```
   TPoligono Poligono2 = IniciaPoligono();
   RellenaPoligono (Poligono2, triangulo, 3);
   PintaPoligono ("Poligono 2", Poligono2);

   TPoligono Poligono3 = IniciaPoligono();
   RellenaPoligono (Poligono3, cuadrado, 3);
   PintaPoligono ("Poligono 3", Poligono3);

   // Aniadir un punto a un poligono es sencillo:
   TPunto2D PuntoNuevo = {0, 10};
   AniadePunto (Poligono3, PuntoNuevo);
   PintaPoligono ("Poligono 3 (actualizado)", Poligono3);

   // Aniadimos dos nuevos puntos
   PuntoNuevo.x = -5;
   PuntoNuevo.y =  5;
   AniadePunto (Poligono3, PuntoNuevo);
```



```
   PuntoNuevo.x = -5;
   PuntoNuevo.y =  0;
   AniadePunto (Poligono3, PuntoNuevo);

   PintaPoligono ("Poligono 3 (actualizado 2)", Poligono3);

   // Desde la nada ....
   TPoligono Poligono4 = IniciaPoligono();

   PuntoNuevo.x = 100;
   PuntoNuevo.y = 100;
   AniadePunto (Poligono4, PuntoNuevo);
   PuntoNuevo.x = 200;
   PuntoNuevo.y = 200;
   AniadePunto (Poligono4, PuntoNuevo);

   PintaPoligono ("Poligono 4", Poligono4);
```

```
   // Poligono vacio

   TPoligono p = IniciaPoligono();
   PintaPoligono ("Poligono p", p);

   return (0);
}

/*****************************************************************/

TPoligono IniciaPoligono (void)
{
   return (0);
}

/*****************************************************************/
```



```
/*****************************************************************/

TNodo * CreaNodo (TPunto2D p)
{
   TNodo * n = new (nothrow) TNodo;
   if (!n) {
      cerr << "\n   Error: no se pudo crear nodo ";
      cerr << "(no hay memoria)\n\n";
      exit (1);
   }
   else {
      n->Punto = p;
      n->sig = 0;
   }
   return (n);
}
/*****************************************************************/
```



```
/*****************************************************************/
void RellenaPoligono (TPoligono & pol, int coordenadas[][2], int NumLados)
{
   TPunto2D UnPunto;
   TNodo * pn;
   TNodo * fin;

   UnPunto.x = coordenadas[0][0];
   UnPunto.y = coordenadas[0][1];
   pn = CreaNodo (UnPunto);
   pol = fin = pn;

   for (int l=1; l<NumLados; l++) {
      UnPunto.x = coordenadas[l][0];
      UnPunto.y = coordenadas[l][1];
      pn = CreaNodo (UnPunto);
```



```
      fin->sig = pn;
      fin = pn;
   }
}
/*****************************************************************/

void PintaPoligono (const char * msg, TPoligono pol)
{
   TNodo * n;
   int c = 0;
   cout << "\n" << msg << ": \n";
   for (n = pol; n != 0; n = n->sig, c++) {
      cout << "   (" << setw(3)<< (n->Punto).x;
      cout << "," << setw(3) << (n->Punto).y << ")\n";
   }
   cout << "   -->" << setw(3) << c << " vertices.\n";
}
```

```
/*************************************************************/
void AniadePunto (TPoligono & pol, TPunto2D punto)
{
   TNodo * pn;
   TPunto2D UnPunto;

   UnPunto.x = punto.x;
   UnPunto.y = punto.y;
   pn = CreaNodo (UnPunto);
   if (pol == 0) pol = pn;
   else {
      TNodo * n;
      for (n = pol; n->sig != 0; n = n->sig);
      n->sig = pn;
   }
}
/*************************************************************/
```


1.5 Matrices dinámicas

Problema:

Gestionar matrices 2D de forma dinámica, en tiempo de ejecución.

Motivación:

El lenguaje proporciona matrices estáticas para las que debe conocerse el tamaño en tiempo de compilación. En el caso de matrices 2D, el compilador debe conocer el número de filas y columnas.

Necesitamos poder crear y trabajar con matrices 2D cuyo tamaño (filas y columnas) sea exactamente el que requiera el problema a resolver.

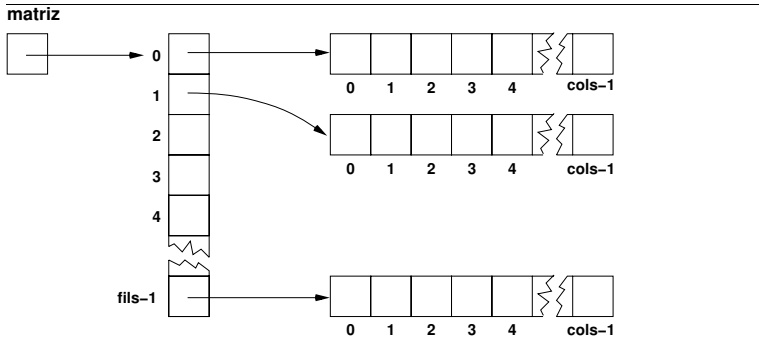
En particular, posibilitaremos:

- Creación de matrices dinámicas.
- Destrucción de matrices dinámicas.
- Acceso mediante índices.

Aproximaciones:

1. Datos guardados en filas independientes.
2. Datos guardados en una única fila.

Solución 1




```
int ** Matriz2D_1 (int fils, int cols);
void LiberaMatriz2D_1 (int **matriz, int fils, int cols);
```

Creación (1)

```
int ** Matriz2D_1 (int fils, int cols)
{
   bool error = false;
   int ** matriz;
   int f, i;

   // "matriz" apunta a un  vect. de punt. a filas

   matriz = new (nothrow) int * [fils];
   if (!matriz) {
      cerr << "Error en reserva (1)"<< endl;
      matriz = 0;
   }

   else {  // Se ha creado el vector de punt. a filas
```



```
   for (f=0; (f<fils) && !error; f++) {

      // "matriz[f]" apuntara a un vector de int

      matriz[f] = new (nothrow) int [cols];

      if (!matriz[f]) {
         cerr << "Error en reserva (2)" << endl;
         error = true; // Detiene el ciclo for

         // Liberar la memoria ya reservada
         for (i=f-1; i>=0; i--) delete [] matriz[i];
         delete [] matriz;

         matriz = 0; // Puntero nulo
      } // if (!matriz[f])
   } // for f
```

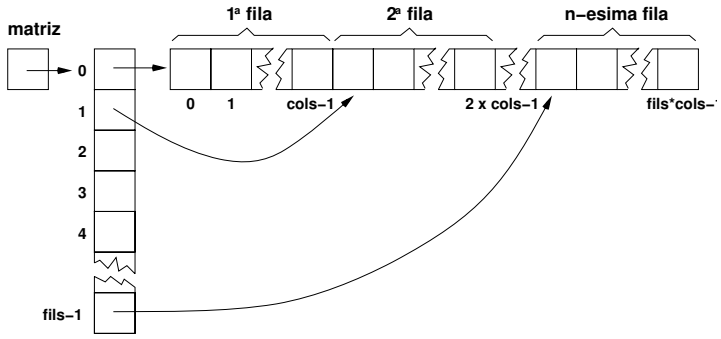


```
   } // else de if (!matriz)

  return (matriz);
}
```

Liberación (1)

```
void LiberaMatriz2D_1 (int **matriz,int fils,int cols)
{
   for (int f=0; f<fils; f++)
      delete [] matriz[f];

   delete [] matriz;
}
```


Solución 2




```
int ** Matriz2D_2 (int fils, int cols);
void LiberaMatriz2D_2 (int **matriz, int fils, int cols);
```

Creación (2)

```
int ** Matriz2D_2 (int fils, int cols)
{
   int ** matriz;
   int f;

   // "matriz" apunta a un  vect. de punt.
   // que apuntaran al inicio de cada fila

   matriz = new (nothrow) int * [fils];

   if (!matriz) {
      cerr << "Error en reserva (3)" << endl;
      matriz = 0; // Puntero nulo
   }
```



```
   else {

     // "matriz[0]" apunta a una sola fila

      matriz[0] = new (nothrow) int [fils*cols];

      if (!matriz[0]) {
         cerr << "Error en reserva (4)" << endl;
         delete [] matriz;
         matriz = 0; // Puntero nulo
      }
      else
         for(f=1; f<fils ; f++)
            matriz[f] = matriz[f-1] + cols;
   }
   return (matriz);
}
```


Liberación (2)

```
void LiberaMatriz2D_2 (int **matriz,
                       int fils, int cols)
{
   delete [] matriz[0];
   delete [] matriz;
}
```

Ejemplo de utilización

```
#include <iostream>
#include <iomanip>
#include <new>
```



```
using namespace std;

void   LeeDimensiones (int &num_filas, int &num_cols);
int ** Matriz2D_1 (int fils, int cols);
int ** Matriz2D_2 (int fils, int cols);
void   PintaMatriz2D (int **matriz, int fils, int cols);
void   LiberaMatriz2D_1 (int **matriz, int fils, int cols);
void   LiberaMatriz2D_2 (int **matriz, int fils, int cols);

/*************************************************************************/

int main (void)
{
   int ** m1; // "m1" y "m2" seran matrices
   int ** m2; // dinamicas 2D.
```

```
   int filas, cols;
   int f, c;

   // Leer num. de filas y columnas

   LeeDimensiones (filas, cols);

   // Crear matrices dinamicas

   cout << "Creando Matriz 1 ("<< filas << "X"<< cols << ")" << endl;

   m1 = Matriz2D_1 (filas, cols);
   if (!m1) {
      cerr << "No se pudo alojar matriz." << endl;
      exit (1);
   }
```



```
   cout << "Creando Matriz 2 ("<< filas << "X"<< cols << ")" << endl;

   m2 = Matriz2D_2 (filas, cols);
   if (!m2) {
      cerr << "No se pudo alojar matriz." << endl;
      exit (2);
   }

   // Rellenarlas (observar el acceso por indices)

   cout << endl << "Rellenando matrices" << endl;

   for (f=0; f<filas; f++)
      for (c=0; c<cols; c++) {
         m1[f][c] = ((f+1)*10)+c+1;
         m2[f][c] = ((f+1)*10)+c+1;
      }
```



```
   // Mostrar su contenido

   cout << endl << "Matriz 1:" << endl;
   PintaMatriz2D (m1, filas, cols);

   cout << endl << "Matriz 2:" << endl;
   PintaMatriz2D (m2, filas, cols);

   // Liberar la memoria ocupada

   cout << endl << "Liberando matriz 1" << endl;
   LiberaMatriz2D_1 (m1, filas, cols);
   cout << "Liberando matriz 2" << endl << endl;
   LiberaMatriz2D_2 (m2, filas, cols);

   return (0);
}
```



```
/*************************************************************************/
void  LeeDimensiones (int &num_filas, int &num_cols) {
   cout << "Numero de filas : ";
   cin >> num_filas;
   cout << "Numero de columnas : ";
   cin >> num_cols;
}
/*************************************************************************/
void PintaMatriz2D (int **matriz, fils, int cols) {
   for (int f=0; f<fils; f++) {
      for (int c=0; c<cols; c++) cout << setw(4) << matriz[f][c];
      cout << endl;
   }
}
/*************************************************************************/

// Matriz2D_1(), Matriz2D_2(), LiberaMatriz2D_1 y LiberaMatriz2D_2()
```

1.6 Ejemplo: Prototipos

Problema:

Un **prototipo** (también llamado *ejemplo*) es un vector d-dimensional (**patrón**) que tiene asociado una etiqueta numérica que indica la clase a la que pertenece. Se trata de calcular y mostrar la media de cada clase a partir de un conjunto de prototipos almacenado en un fichero de texto.

Análisis

Entradas:

El programa recibirá N prototipos de dimensión d que pertenecen a J clases. Los prototipos se proporcionan en un *fichero de prototipos* cuyas especificaciones son:

1. Es un fichero de texto.
2. Cada línea contiene un único prototipo y cada prototipo se encuentra en una sola línea $\Rightarrow$ *el fichero contendrá tantas líneas como prototipos*.
3. No se impone ningún orden entre los prototipos.

4. Si los prototipos son de dimensión d, cada línea está formada por $d+1$ columnas:
   - *a*) Las primeras d columnas son valores *reales* y corresponden a los valores de los d atributos.
   - *b*) La última columna contiene un valor *entero* que indica su *clase*.
   - *c*) Las columnas están separadas por un número indeterminado de espacios en blanco.
5. El conjunto de etiquetas es completo, siendo la menor 1. Así, para un problema en el que $J = 3$, por ejemplo, existen prototipos de clase 1, 2 y 3 únicamente.

Se supone que el fichero de prototipos es correcto.

Ejemplo:

- 2 clases: **1** (Mujer) y **2** (Hombre).
- 2 atributos: 1 (Peso) y 2 (Altura).

```
78.4   175   2
66.3   170   1
60.4   162   1
72.4   171   2
   ........
```

En definitiva, el programa requiere conocer:

1. El nombre del fichero de prototipos (cadena).
2. El número de prototipos, N (entero).
3. El número de atributos, d (entero).
4. El número de clases, J (entero).

Nota: Los valores de N, d y J pueden calcularse examinando el fichero de entrada, aunque en este momento no se dispone de suficientes conocimientos.

Salidas:

El programa debe mostrar los valores medios de las J clases. Como los datos de entrada son de dimensión d, se mostrará para cada una de las J clases, d valores: media del atributo 1, . . . , media del atributo d.

Los valores medios calculados deben ser reales.

Relación entre Entradas y Salidas:

La media del atributo i para la clase j, m_{j_i}, se calcula de manera trivial:

$$m_{j_i} = \frac{1}{N_j} \sum_{k=1}^{N_j} {p^k}_{j_i}$$

donde:

- N_j es el número de prototipos de la clase j.
- ${p^k}_j$ es el prototipo k-ésimo de la clase j.
- ${p^k}_{j_i}$ es el valor del atributo i del prototipo k-ésimo de la clase j.

Diseño

Tipos de datos:

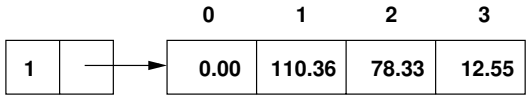
El número de prototipos y dimensión de los mismos es desconocido *a priori* por lo que los prototipos se guardarán en memoria dinámica.

Prototipo.

La representación de un prototipo requiere:

- La etiqueta de la clase (`int`)
- Los valores de los atributos (vector de d valores `float`). El valor de d es, *a priori*, desconocido $\Rightarrow$ *memoria dinámica*.


```
struct prototipo {

  float *atributos; // Valores de los atrs.
  int clase;        // Etiqueta de clase

};
```



Nota: Índices `0..d` (`d`+1 casillas)

Conjunto de prototipos.

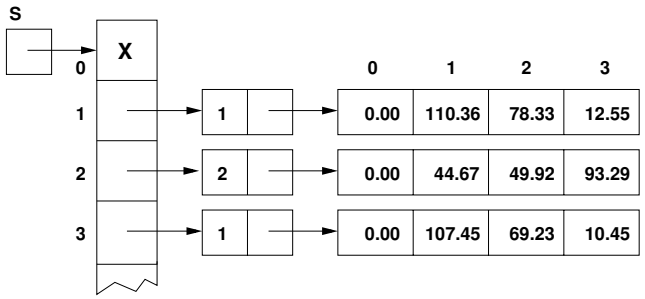
- El número de prototipos es, *a priori*, desconocido ⇒ *memoria dinámica*.

  Usaremos un vector (dinámico) de punteros a `prototipo`.

- Para facilitar el paso de parámetros, añadimos una nueva referencia:

```
typedef prototipo ** CtoProt;
```


Por ejemplo, si `S` se declara de tipo `CtoProt` y se inicializa adecuadamente:



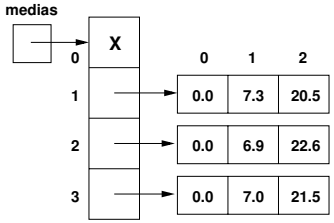
Nota: Índices `0..N` (`N`+1 casillas)

Medias.

Se necesitan $J \times d$ datos `float`, pero:

El número de clases, J, y la dimensionalidad de los datos, d, es, a priori, desconocido.

Consecuencia: matriz con J filas y d columnas que resida en *memoria dinámica*.



Algoritmo:

Algoritmo básico.

1. Obtener entradas: nombre del fichero de prototipos y los parámetros asociados (N, J y d).
2. Copiar los prototipos del fichero en memoria.
3. Calcular las medias.
4. Mostrar las medias.

Refinamiento.

⋆ Los parámetros N, J y d dependen del fichero, por lo que su entrada puede hacerse en dos pasos:

1.1 Obtener nombre del fichero de prototipos.
Se obtiene de `cin`.
Previsión 1: *tomarlo de la línea de órdenes*.

1.2 Obtener los parámetros asociados (N, J y d).
Se obtienen de `cin`.
Previsión 2: *calcularlos automáticamente*.

⋆ El programa debería mostrar más información para asegurar que funciona correctamente:

1.3 Mostrar los valores de entrada.
2.1 Copiar los prototipos del fichero en memoria.
2.2 Mostrar el conjunto de prototipos.

⋆ Conjunto de prototipos y medias en memoria dinámica ⇒ liberarla antes de acabar.

Algoritmo refinado.

1. Obtener nombre del fichero de prototipos.
2. Obtener los parámetros asociados (N, J y d).
3. Mostrar los valores de entrada.
4. Copiar los prototipos del fichero en memoria.
5. Mostrar el contenido del conjunto de prototipos.
6. Calcular las medias.
7. Mostrar las medias.
8. Liberar la memoria ocupada.

Descomposición modular:

En primera instancia: un módulo para cada paso.

1. Obtener nombre del fichero de prototipos.

```
void ObtenerNombreFichero (string & nbre);
```

   Devuelve (ref.) una cadena: el nombre del fichero de prototipos.

2. Obtener los parámetros asociados (N, J y d).

```
void ObtenerDatos (string nombre, int &N, int &J, int &d);
```

   Devuelve (ref.) los valores de `N`, `J` y `d` asociados al fichero `nombre`.

3. Mostrar los valores de entrada.

```
void MostrarDatos (string nombre, int N, int J, int d);
```

Muestra en cout los valores de N, J y d asociados al fichero nombre.

4. Copiar los prototipos del fichero en memoria.

```
CtoProt RellenaPrototipos (string nombre, int N, int d)
```

Crea un cto. de N prototipos de dimensión d y lo inicializa a partir del fichero nombre.
Devuelve el conjunto creado e inicializado.
No se requiere reserva previa de memoria.

5. Mostrar el contenido del conjunto de prototipos.

```
void PintaPrototipos (CtoPrtot prot, int N, int d)
```

Muestra el contenido del cto. prot, compuesto por N prototipos de dimensión d.

6. Calcular las medias.

```
float ** CreaYCalculaMedias (int J, int d, int N, CtoProt prot)
```

Crea una matriz dinámica bidimensional (J filas y d columnas) y la inicializa calculando las medias del conjunto prot, compuesto por N prototipos de dimensión d de J clases.
Devuelve la matriz dinámica bidimensional creada e inicializada.
No se requiere reserva previa de memoria.

7. Mostrar las medias.

```
void PintaMedias (int J,int d,float **mu)
```

Muestra los valores guardados en la matriz dinámica bidimensional mu, que tiene J filas y d columnas.

8. Liberar la memoria ocupada.

La liberación de memoria involucra a dos módulos:

```
void LiberaPrototipos (CtoProt prot,int N);
```

Libera la memoria ocupada por el conjunto prot, que tiene N prototipos.

```
void LiberaMedias (int J, float **mu);
```

Libera la memoria ocupada por la matriz mu, que tiene J filas.

9. Módulos adicionales:

```
CtoProt ReservaPrototipos (int N, int d);
```

Reserva memoria para alojar a N prototipos de dimensión d.
Devuelve el conjunto reservado, pero no inicializado.
Llamado por RellenaPrototipos().

```
float ** ReservaMatriz2D (int fils,int cols)
```

Reserva memoria para alojar una matriz dinámica bidimensional de float que tiene fils filas y cols columnas.
Devuelve la matriz reservada, pero no inicializada.
Llamada por CreaYCalculaMedias().

`void LiberaMatriz2D (float **matriz, int fils,int cols)`

Libera la memoria reservada para la matriz dinámica bidimensional de `float` llamada `matriz`, para la que se reserva espacio para `fils` filas y `cols` columnas.
Llamada por `LiberaMedias()`.

El separar la reserva de la inicialización tiene la ventaja de que si se requiere otra función que necesite reservar memoria para un conjunto de prototipos (p.e., hacer una copia en otro conjunto, extraer a un nuevo conjunto los de una clase dada, etc.) puede *reutilizarse*.

El mismo comentario puede aplicarse a las funciones que crean y liberan matrices dinámicas 2D.

Modularidad a nivel de ficheros:

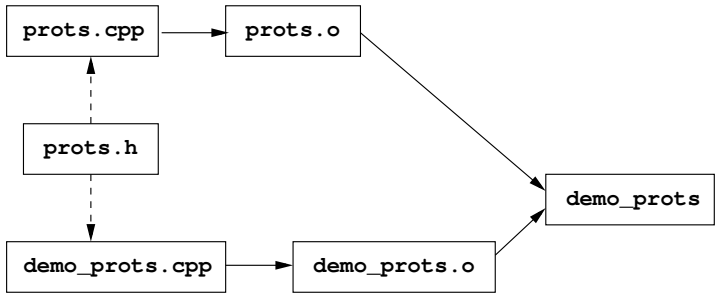
Modelo de compilación separada de C/C++

- `prots.h`
  - Definición de los tipos de datos ofertados.
  - Declaración de las funciones ofertadas.
- `prots.cpp`
  - Implementación de las funciones ofertadas.
  - Implementación de las funciones privadas. En este caso, únicamente `ReservaPrototipos()`.
- `demo_prots.cpp`

  Contiene únicamente la función `main()`.

- `Makefile`

  Especificación de dependencias entre ficheros para la creación del ejecutable.



Codificación

`prots.h`

```
#ifndef PROTS
#define PROTS

// Definicion de los tipos "prototipo" y "CtoProt"

   struct prototipo {
      float *atributos; // Valores de los atributos
      int clase;        // Etiqueta de clase
   };
   typedef prototipo ** CtoProt;

// Funciones para obtener y mostrar los valores de los
// parametros asociados al fichero de prototipos
```

```
void ObtenerNombreFichero (string & nbre);
void ObtenerDatos (string nombre,int &N,int &J,int &d);
void MostrarDatos (string nombre,int N,int J,int d);

// Funciones para cargar/liberar prototipos en memoria y su procesamiento.

CtoProt RellenaPrototipos (string nombre,int N,int d);
void LiberaPrototipos (CtoProt prot, int N);
void PintaPrototipos (CtoProt prot, int N, int d);

// Funciones asociadas a medias.

float ** CreaYCalculaMedias (int J, int d, int N, CtoProt prot);
void PintaMedias (int J, int d, float **media);
void LiberaMedias (float **media, int J, int d);

#endif
```


prots.cpp

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
#include <new>
using namespace std;

#include "prots.h"

// Funciones privadas:

CtoProt  ReservaPrototipos (int N, int d);
float ** ReservaMatriz2D (int fils, int cols);
void     LiberaMatriz2D  (float **matriz, int fils, int cols);

// ********************************************************************
```



```
void ObtenerNombreFichero (string & nbre)
{
   cout << endl << "Nombre del fichero de prototipos: ";
   cin >> nbre;
}
// ********************************************************************
void ObtenerDatos (string nombre,int &N,int &J,int &d)
{
   cout << "Introducir los valores de los parametros ";
   cout << "asociados al fichero: "<< nombre << endl;
   cout << "   Num. de prototipos: ";
   cin >> N;
   cout << "   Num. de clases: ";
   cin >> J;
   cout << "   Num. de atributos: ";
   cin >> d;
}
```



```
void MostrarDatos (string nombre, int N, int J, int d)
{
   cout << "Fichero: "<< nombre << endl;
   cout << "   Num. de prototipos: " << N << endl;
   cout << "   Num. de clases: " << J << endl;
   cout << "   Num. de atributos: " << d << endl;
}
// ********************************************************************

CtoProt ReservaPrototipos (int N, int d)
{
   CtoProt  prot;
   prot = new (nothrow) prototipo * [N+1];
   if (!prot) {
      cerr << "No hay mem. para prots. (1)" << endl;
      exit (1);
   }
```

```
   prot[0] = 0;  // Puntero nulo

   for (int p=1; p<=N; p++) {
      prot[p] = new (nothrow) prototipo;
      if (!prot[p]) {
         cerr << "No hay mem. para prots. (2)"<<endl;
         exit (2);
      }
      (prot[p])->atributos = new (nothrow) float [d+1];
         if (!(prot[p])->atributos) {
            cerr << "No hay mem. para prots. (3)"<<endl;
            exit (3);
         }
   } // for p

   return (prot);
}
```



```
CtoProt  RellenaPrototipos (string nombre,int N,int d)
{
   CtoProt  prot;
   ifstream fprot;
   int   clase;
   float valor;

   // Reservar memoria para "N" prots. de dim. "d".

   prot = ReservaPrototipos (N, d);

   // Rellenar "prot" a partir del fichero

   fprot.open (nombre.c_str(), ios::in);

   for (int p=1; p<=N; p++) {
```



```
      (prot[p]->atributos)[0] = 0.0;
      for (int atr=1; atr<=d; atr++) {
         fprot >> valor;
         (prot[p]->atributos)[atr] = valor;
      }

      fprot >> clase;
      prot[p]->clase = clase;

   } // for p

   fprot.close();

   return (prot);
}
```



```
void LiberaPrototipos (CtoProt prot, int N)
{
   for (int p=1; p<=N; p++) {
      delete [] ((prot[p])->atributos);
      delete prot[p];
   }
   delete [] prot;
}

// ********************************************************************

void  PintaPrototipos (CtoProt prot, int N, int d)
{
   cout << endl << endl;
   cout << "Prototipos leidos y guardados:" << endl;
```

```
   cout.setf (ios::fixed);
   cout.setf (ios::showpoint);
   cout.precision (2);

   for (int p=1; p<=N; p++) {

      cout << "Prot. "<< setw(3) << p <<": ";
      for (int atr=1; atr<=d; atr++) {
         cout<<setw(6)<<(prot[p]->atributos)[atr]<<"  ";
      }

      cout << "["<<prot[p]->clase<<"]"<<endl;

   } // for p
}

// ****************************************************************
```



```
float ** CreaYCalculaMedias (int J, int d, int N, CtoProt prot)
{
   float **mu, *patron;
   int    *cont;
   float   valor;

   cont = new (nothrow) int [J+1];
   if (!cont) {
      cerr << "No hay mem. para contadores (1)"<< endl;
      exit (4);
   }
   mu = ReservaMatriz2D (J+1, d+1);
   if (!mu) {
      cerr << "No hay memoria para medias" << endl;
      exit (2);
   }
   mu[0] = 0; // Puntero nulo
```



```
   // Inicializacion
   for (int j=1; j<=J; j++)
      for (int atr=0; atr<=d; atr++) {
         mu[j][atr] = 0.0;
         cont[j] = 0;
      }
   // Calculo

   for (int p=1; p<=N; p++) {
      for (int atr=1; atr<=d; atr++) {
         patron = prot[p]->atributos;
         mu[prot[p]->clase][atr] += patron[atr];
      }
      cont[prot[p]->clase]++;
   } // for p
```



```
   for (int j=1; j<=J; j++)
      for (atr=1; atr<=d; atr++)
         mu[j][atr] /= cont[j];

   delete [] cont;
   return (mu);
}

// ****************************************************************
void PintaMedias (int J, int d, float **mu)
{
   cout << endl << endl;
   cout <<"Valores medios (por clase):"<< endl<< endl;

   cout.setf (ios::fixed);
   cout.setf (ios::showpoint);
   cout.precision (2);
```

```
   for (int j=1; j<=J; j++) {
      cout << "Clase " << setw(3) << j << ": ";
      for (int atr=1; atr<=d; atr++) {
         cout << setw(8) << mu[j][atr];
      }
      cout << endl;
   }
   cout << endl;
}

// *****************************************************************

void LiberaMedias (float **mu, int J, int d)
{
   LiberaMatriz2D (mu, J+1, d+1);
}
```



```
float ** ReservaMatriz2D (int fils, int cols)
{
   bool error = false;
   float ** matriz;

   // "matriz" apunta a un  vect. de punt. a filas

   matriz = new float * [fils];
   if (!matriz) {
      cerr << "Error en reserva (1)"<< endl;
      matriz = 0;
   }
   else {  // Se ha creado el vector de punt. a filas
      for (int f=0; (f<fils) && !error; f++) {

         // "matriz[f]" apuntara a un vector de int
         matriz[f] = new float [cols];
```



```
         if (!matriz[f]) {
            cerr << "Error en reserva (2)"<<endl;
            error = true; // Detiene el ciclo for

            // Libera la memoria ya reservada

            for (int i=f-1; i>=0; i--)
               delete [] matriz[i];
            delete [] matriz;

            matriz = 0; // Puntero nulo
         }
      } // for f
   }
  return (matriz);
}
```



```
// *****************************************************************

void LiberaMatriz2D (float **matriz,int fils,int cols)
{
   for (int f=0; f<fils; f++)
      delete [] matriz[f];

   delete [] matriz;
}
// *****************************************************************
```

demo_prots.cpp

```
#include <string>

using namespace std;

#include "prots.h"

int main (void)
{
   string nombre;   // Nombre del fichero de prototipos
   CtoProt prot;    // ED para guardar prototipos
   int N;           // Num. de prototipos
   int J;           // Num. de clases
   int d;           // Num. de atributos

   float ** medias; // ED para calcular y guardar las medias (por clase)
```



```
   // Obtener (de "cin") el nombre del fichero.
   // Mas adelante se tomara de la linea de ordenes.

   ObtenerNombreFichero (nombre);

   // Obtener los parametros asociados al fichero.
   // En una version futura el calculo es automatico

   ObtenerDatos (nombre, N, J, d );

   // Mostrar (en "cout") los parametros asociados al fichero.

   MostrarDatos (nombre, N, J, d );

   // Crear y rellenar la ED "prot" a partir del fichero de prototipos.

   prot = RellenaPrototipos (nombre, N, d);
```



```
   // Mostrar los datos leidos y guardados en "prot"

   PintaPrototipos (prot, N, d);

   // Calcular y mostrar las medias (por clase)

   medias = CreaYCalculaMedias (J, d, N, prot);
   PintaMedias (J, d, medias);

   // Liberar la memoria ocupada

   LiberaPrototipos (prot, N);
   LiberaMedias (medias, J, d);

   return (0);
}
```


Makefile

```
SOURCE = source
BIN = bin
INCLUDE = include
OBJ = obj

all : prots.o demo_prots.o demo_prots

prots.o : $(SOURCE)/prots.cpp $(INCLUDE)/prots.h
        g++ -c -o $(OBJ)/prots.o -I$(INCLUDE) $(SOURCE)/prots.cpp

demo_prots.o : $(SOURCE)/demo_prots.cpp $(INCLUDE)/prots.h
        g++ -c -o $(OBJ)/demo_prots.o -I$(INCLUDE) $(SOURCE)/demo_prots.cpp

demo_prots : $(OBJ)/demo_prots.o $(OBJ)/prots.o
        g++ -o $(BIN)/demo_prots $(OBJ)/demo_prots.o $(OBJ)/prots.o
```

Pruebas

- Suposición **básica**:

  El fichero de prototipos es **correcto**.

- Pruebas sobre los valores de entrada:

1. Es un fichero de prototipos (`.prot`)
   Proporcionar nombres sin extensión `.prot`

2. Existe el fichero de prototipos
   Probar nombres de ficheros que no existen.

3. Son coherentes los valores de N, J y d con el contenido del fichero
   Nota: N, J y d se calcularán automáticamente $\Rightarrow$ esta prueba no se realizará.

- Pruebas de cálculo.

1. Prueba de valores extremos:
   Media de una clase que tiene un único prototipo.

Resultados de las pruebas.

1. Sobre el nombre del fichero de prototipos.

   `ObtenerNombreFichero()` no comprueba nada: acepta *cualquier* nombre.

2. Sobre la existencia del fichero de prototipos.

   No se realiza ninguna comprobación de la existencia del fichero. `RellenaPrototipos()` *abre* un fichero que no existe e intenta leer de él: errores.

   Esta comprobación puede hacerse en la función `ObtenerNombreFichero()`, una vez que se ha validado el nombre proporcionado.

3. Prueba de cálculo.

   No se detectó ningún problema.

Conclusiones.

1. Validar el nombre en `ObtenerNombreFichero()`.

2. Comprobar la existencia del fichero en `ObtenerNombreFichero()`.

3. Escribir un módulo específico para comprobar la existencia de un fichero:

   ```
   bool ExisteFichero (string & nbre);
   ```

   Comprueba la existencia del fichero llamado `nombre` y su disponibilidad para ser abierto para lectura.

   Este módulo puede ser privado para `prots.cpp`

Codificación (2)

`prots.h`

¡¡No hay cambios!!

`prots.cpp`

Añadir, a la lista de funciones privadas:

```
bool ExisteFichero (string & nbre);
```

Sustituir el módulo `ObtenerNombreFichero()`:


```
void ObtenerNombreFichero (string & nbre)
{
   int pos; // Posicion de ".prot" en "nbre"

   cout << endl << "Nombre del fichero de prototipos: ";
   cin >> nbre;
   cout << endl;

   // Filtro de entrada para admitir unicamente nombres terminados en ".prot"

   pos = nbre.find (".prot");
   if ((pos < 0) || ((pos + 5) != nbre.length())) {
      cerr << "Error: Nombre de fichero incorrecto ";
      cerr << "(debe tener extension .prot)" << endl;
      cerr << endl;
      exit (1);
   }
```



```
   // Comprobar la existencia del fichero.

   if (!ExisteFichero (nbre)) {
      cerr << "Error: El fichero " << nbre;
      cerr << " no puede abrirse." << endl << endl;
      exit (1);
   }
}
```

Añadir, al final del fichero, la definicin de `ExisteFichero ()`:

```
bool ExisteFichero (string &nombre)
{
   ifstream fichero;
   bool problema;
```



```
   fichero.open (nombre.c_str());

   problema = fichero.fail();

   if (!problema) fichero.close();

   return ((problema) ? false : true);
}
```

`demo_prots.cpp`

No hay cambios!!