Relación de ejercicios tema 3: punteros y gestión dinámica de memoria

Metodología de la programación, 2022-2023

Contenido:

1 Problemas básicos con punteros **1**
- 1.1 Uso de punteros . . . 1
- 1.2 Aritmética de punteros: punteros y arrays . . . 2

2 Punteros y funciones **3**
- 2.1 Funciones que reciben arrays . . . 3
- 2.2 Punteros a funciones . . . 4

3 Punteros, struct y class **5**

4 Memoria dinámica **7**
- 4.1 Vectores dinámicos . . . 8
- 4.2 Matrices dinámicas . . . 13
- 4.3 Listas de celdas enlazadas . . . 16

1 Problemas básicos con punteros

1.1 Uso de punteros

1. Describa la salida de los siguientes programas:

a)
```
#include <iostream>
using namespace std;

int main (){
  int a = 5, *p;

  a = *p * a;
  if (a == *p)
    cout << "a es igual a *p" << endl;
  else
    cout << "a es diferente a *p" << endl;
  return 0;
}
```

b)
```
#include <iostream>
using namespace std;

int main (){
  int a = 5, *p;

  *p = *p * a;
  if (a == *p)
    cout << "a es igual a *p" << endl;
  else
    cout << "a es diferente a *p" << endl;
  return 0;
}
```

c)
```
#include <iostream>
using namespace std;

int main (){
  int a = 5, *p = &a;

  *p = *p * a;
  if (a == *p)
    cout << "a es igual a *p" << endl;
  else
    cout << "a es diferente a *p" << endl;
  return 0;
}
```

d)
```
#include <iostream>
using namespace std;

int main (){
  int a = 5, *p = &a, **p2 = &p;

  **p2 = *p + (**p2 / a);
  *p = a+1;
  a = **p2 / 2;
  cout << "a es igual a: " << a << endl;
  return 0;
}
```

2. Implementa una función que reciba un puntero a entero y que:

   (a) Eleve al cuadrado el dato apuntado.

   (b) Ponga a cero el puntero.

   Tal función podría usarse de la siguiente forma:

```
int a=6;
int *q;
q = &a;
elevarAlCuadrado(q);
cout << a << q; // Debería salir 36 0
```

3. Indica qué ocurre en las siguientes situaciones:

   (a)
```
int *p;
const int a=2;
p = &a;
```

   (b)
```
const int *p;
int a;
p = &a;
*p = 7;
a = 8;
```

   (c)
```
int entero=10;
const int enteroConst=20;
int *v1;
const int* v3;
v1=&entero;
v3=&enteroConst;
```

```
int* const v2=v1;
const int* const v4=v3;
v1 = v2;
v1 = v3;
v1 = v4;
```

```
int* const v2=v3;
```

```
const int* const v4=v3;
int* const v2=v4;
```

```
int* const v2=v1;
v3 = v1;
v3 = v2;
v3 = v4;
```

```
const int* const v4=v1;
```

```
int* const v2=v1;
const int* const v4=v2;
```

```
int* const v2=v1;
const int* const v4=v3;
*v1=*v2;
*v1=*v3;
*v1=*v4;
```

```
int* const v2=v1;
const int* const v4=v3;
*v2=*v1;
*v2=*v3;
*v2=*v4;
```

```
int* const v2=v1;
const int* const v4=v3;
*v3=*v1;
*v3=*v2;
*v3=*v4;
```

```
int* const v2=v1;
const int* const v4=v3;
*v4=*v1;
*v4=*v2;
*v4=*v3;
```

1.2 Aritmética de punteros: punteros y arrays

4. Declare una variable `v` como un array de 1000 enteros. Escriba un trozo de código que recorra el array y modique todos los enteros negativos cambiándolos de signo. No se permite usar el operador `[]`, es decir, el recorrido se efectuará usando aritmética de punteros y el bucle se controlará mediante un contador entero.

5. Modifique el código del problema anterior para controlar el final del bucle con un puntero a la posición siguiente a la última.

6. Dado un array de 10 elementos, haz un bucle que busque el máximo y el mínimo (sin usar el operador `[]`). Al acabar el bucle tendremos un puntero apuntando a cada uno de ellos.

2 Punteros y funciones

2.1 Funciones que reciben arrays

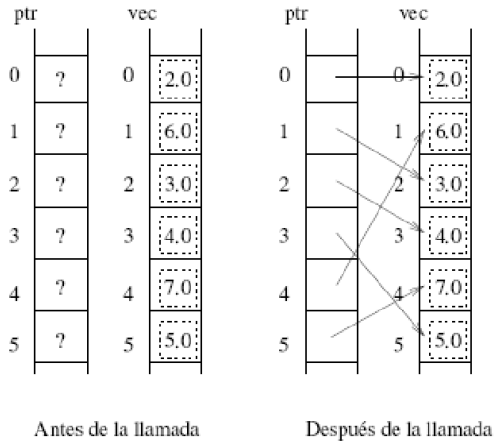
7. Implemente las siguientes funciones sobre cadenas de caracteres estilo C usando aritmética de punteros (sin usar el operador `[]`):

   (a) Función `compararCadenas` que compara dos cadenas. Devuelve un valor negativo si la primera es más *pequeña*, positivo si es más *grande* y cero si son *iguales*.

   (b) Función `insertarCadena` que inserte una cadena dentro de otra, en una posición dada. Se supone que hay suficiente memoria en la cadena de destino.

   Se supone que no es necesario pasar el tamaño de las cadenas (recordad que el carácter nulo delimita el final de la cadena).

8. Escriba una función `ordenacionPorBurbuja()` que reciba como entrada un array de números junto con su longitud y que nos devuelva un array de punteros a los elementos del array de entrada de forma que los elementos apuntados por dicho array de punteros estén ordenados (véase la siguiente figura).



Debe usarse el algoritmo de ordenación de la burbuja. Este algoritmo consiste en hacer una comparación entre cada dos elementos consecutivos desde un extremo del vector hasta el otro, de forma que el elemento de ese extremo queda situado en su posición final. Una posible implementación es la siguiente:

```
void ordenacionBurbuja (int vec[], int n) {
   for (int i=n-1; i>0; --i) {
      for (int j=0; j<i; ++j) {
         if (vec[j] > vec[j+1]) {
            int aux = vec[j];
            vec[j] = vec[j+1];
            vec[j+1]= aux;
         }
      }
   }
}
```

Note que el array de punteros debe ser un parámetro de la función, y estar reservado previamente a la llamada con un tamaño, al menos, igual al del array. Una vez escrita la función, considere la siguiente declaración:

```
int vec [1000];
int *ptr [1000];
```

y escriba un trozo de código que, haciendo uso de la función, permita:

(a) Ordenando punteros, mostrar los elementos del array, ordenados.

(b) Ordenando punteros, mostrar los elementos de la segunda mitad del array, ordenados.

sin modificar el array de datos `vec`.

2.2 Punteros a funciones

9. El algoritmo de ordenación del ejercicio anterior nos ordena el array en orden creciente. La ordenación en orden decreciente se obtendría si cambiamos en la comparación del `if` , el operador `>` por el operador `<`. Para crear un algoritmo genérico podemos introducir un nuevo parámetro a la función `ordenacionPorBurbuja()` que indique el orden de los elementos. En nuestro caso, podemos pasarle la función que indica el orden que hay entre dos enteros, es decir, un parámetro adicional con un puntero a función que calcula el orden. Por ejemplo, podemos crear la siguiente función:

```
int ordenCreciente(int l, int r)
{
   return l-r;
}
```

La función devuelve un número negativo si el primero es menor que el segundo, un positivo si es al contrario, o un cero en caso de que sean iguales. Usando esta función, podemos indicar en el algoritmo de ordenación `ordenacionPorBurbuja()` que queremos un orden de menor a mayor, cambiando la comparación del `if` por:

```
if(ordenCreciente(vec[j], vec[j+1]) > 0)
```

Si queremos obtener un orden de mayor a menor, podemos usar otra función:

```
int ordenDecreciente(int l, int r)
{
  return r-l;
}
```

Ahora tendríamos que poner el `if` de `ordenacionPorBurbuja()` de la siguiente forma:

```
if(ordenDecreciente(vec[j], vec[j+1]) > 0)
```

Añada un nuevo parámetro puntero a función, a la función `ordenacionPorBurbuja()` del ejercicio anterior, según lo explicado más arriba. La idea es permitir que la función `ordenacionPorBurbuja()` pueda usarse para ordenar un array de menor a mayor o de mayor a menor. Si llamamos a `ordenacionPorBurbuja()` con la función `ordenCreciente()`, el array se ordenaría de menor a mayor, y si usamos `ordenDecreciente()`, se ordenaría de mayor a menor.

3 Punteros, struct y class

10. Represente gráficamente la disposición en memoria de las variables del programa mostrado a continuación, e indique lo que escribe la última sentencia de salida.

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

struct Celda
{
    int d ;
    Celda *p1 , *p2 , *p3 ;
};

int main ( int argc , char * argv [ ] )
{
    Celda a, b, c, d;

    a.d = b.d = c.d = d.d = 0 ;

    a.p1 = &c;
    c.p3 = &d;
    a.p2 = a.p1->p3;
    d.p1 = &b;
    a.p3 = c.p3->p1;
    a.p3->p2 = a.p1;
    a.p1->p1 = &a;
    a.p1->p3->p1->p2->p2 = c.p3->p1;
    c.p1->p3->p1 = &b;
    (*((*(c.p3->p1)).p2->p3)).p3 = a.p1->p3;
    d.p2 = b.p2;
    (*(a.p3->p1)).p2->p2->p3 = (*(a.p3->p2)).p3->p1->p2 ;

    a.p1->p2->p2->p1->d = 5;
    d.p1->p3->p1->p2->p1->p1->d = 7 ;
    (*(d.p1->p3)).p3->d = 9 ;
    c.p1->p2->p3->d = a.p1->p2->d - 2 ;
    (*(c.p2->p1)).p2->d = 10 ;

    cout << "a="<<a.d<<" b="<<b.d<<" c="<<c.d<<" d="<<d.d<<endl ;
}
```

11. Represente gráficamente la disposición en memoria de las variables del programa mostrado a continuación, e indique lo que escribe la última sentencia de salida. Tenga en cuenta que el operador `->` tiene más prioridad que el operador `*`.

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

struct SB; // declaración adelantada
struct SC; // declaración adelantada
struct SD; // declaración adelantada

struct SA {
    int dat ;
    SB *p1 ;
} ;

struct SB {
    int dat ;
    SA *p1 ;
    SC *p2 ;
} ;

struct SC {
    SA *p1 ;
    SB *p2 ;
    SD *p3 ;
} ;

struct SD {
    int *p1 ;
    SB *p2 ;
} ;

int main ( int argc , char * argv [ ] )
{
    SA a ;
    SB b ;
    SC c ;
    SD d ;
    int dat ;

    a.dat = b.dat = dat = 0 ;

    a.p1 = &b ;
    b.p1 = &a ;
    b.p2 = &c ;
    c.p1 = b.p1 ;
    c.p2 = &(*(a.p1)) ;
    c.p3 = &d;
    d.p1 = &dat;
    d.p2 = &(*(c.p1)->p1);
    *(d.p1) = 9;
    (*(b.p2)->p1).dat = 1;
    (*((b.p2)->p3->p2)->p1).dat = 7 ;
    *((*((*(c.p3->p2)).p2->p3)).p1) = (*(b.p2)).p1->dat + 5 ;
    cout << "a.dat=" << a.dat << " b.dat=" << b . dat << " dat=" << dat << endl ;
}
```

4 Memoria dinámica

12. Describa la salida del siguiente programa:

```
#include <iostream>
using namespace std;  public:


int main (){
   int *p1, *p2;

   p1 = new int;
   *p1 = 42;
   p2 = p1;
   cout << *p1 << " y " << *p2 << endl;

   *p2 = 53;
   cout << *p1 << " y " << *p2 << endl;

   p1 = new int;
   *p1 = 88;
   cout << *p1 << " y " << *p2 << endl;
}
```

13. Dadas las siguientes declaraciones

```
struct Electrica {
   char corriente[30];
   int voltios;
};

Electrica *p = new Electrica(), *q = new Electrica();
```

se pide averiguar qué hacen cada una de las siguientes sentencias. ¿Hay alguna inválida?

```
a.  strcpy(p->corriente,"ALTERNA");   e.  strcpy(p->corriente,"ALTA");
b.  p->voltios = q->voltios;          f.  p->corriente = q->voltios;
c.  *p = *q;                          g.  p = 54;
d.  p = q;                            h.  *q = p;
```

4.1 Vectores dinámicos

14. Queremos hacer un programa `leerCaracteres.cpp` que lea uno a uno los caracteres de un fichero de texto pasado al programa como argumento de `main()` (o bien de la entrada estándar si no se pasa tal argumento), los cargue en un array dinámico de caracteres y luego lo muestre en la salida estándar. El array dinámico será controlado con un puntero a `char` (`arraychar`) y un entero con el número de caracteres en el array. El array tendrá en todo momento el tamaño justo necesario para contener los caracteres leídos (algo que hace muy ineficiente este programa). La función `main()` se da a continuación.

```
#include <iostream>
#include <fstream> // ifstream
#include "ArrayDinamico.h"

using namespace std;

int main(int argc, char* argv[])
{
   char* arraychar;
   int nCaracteres;

   inicializar(arraychar, nCaracteres);
   if (argc==1)
      leer(cin, arraychar, nCaracteres);
   else {
      ifstream flujo;
      flujo.open(argv[1]);
      if (!flujo) {
         cerr << "Error: Fichero " << argv[1] << " no válido." << endl;
         return 1;
      }
      leer(flujo, arraychar, nCaracteres);
      flujo.close();
   }
   mostrar(cout, arraychar, nCaracteres);
   liberar(arraychar, nCaracteres); // Libera la memoria dinámica reservada
}
```

Como puede verse, el programa puede ser ejecutado de dos formas:

(a) Dando un argumento de entrada a main con el nombre del fichero de texto:

```
prompt> ./leerCaracteres fichero.txt
Este es un fichero de prueba que contiene
solo dos líneas.
```

(b) Leyendo los datos de la entrada estándar:

```
prompt> cat fichero.txt | ./leerCaracteres
Este es un fichero de prueba que contiene
solo dos líneas.
```

Construye un módulo para trabajar con arrays dinámicos de caracteres (escribiendo los ficheros `ArrayDinamicoChar.h` y `ArrayDinamicoChar.cpp`) que contenga las siguientes funciones y pruébalo con el anterior programa para leer un fichero de texto:

(a) `inicializar` para inicializar el array dinámico.

(b) `liberar` para liberar la memoria dinámica ocupada por el array dinámico.

(c) `mostrar` para mostrar en un flujo de salida los caracteres del array dinámico.

(d) `redimensionar` que amplía el tamaño de un array dinámico con un determinado valor entero `incremento` recibido como parámetro.

(e) `aniadir` que añade un carácter al final de un array dinámico, aumentando previamente su tamaño en uno.

(f) `leer` para leer los caracteres uno a uno de un flujo de entrada, y guardarlos en un array dinámico recibido como parámetro.

La función `leer` se da a continuación:

```
void leer(istream& flujo, char* &array, int& nchar){
   char caracter;

   liberar(array, nchar);
   while(flujo.get(caracter)){
      aniadir(array, nchar, caracter);
   }
}
```

(g) Haz una nueva versión de todo el programa (`main` y funciones de los apartados a) hasta f)) para que el array dinámico esté definido con un **tipo de dato abstracto** mediante el siguiente `struct`:

```
struct VectorDinamicoChar{
   char* arrayChar;  // array dinámico de char
   int nCaracteres;  // nº de caracteres actualmente en el array
}
```

El uso de este tipo de dato abstracto tiene la ventaja que podría modificarse en el futuro (añandiendo nuevos datos al struct o cambiado el nombre de algunos de ellos) sin que tuviésemos que cambiar los programas que lo usan.

(h) Haz una nueva versión de todo el programa (`main` y funciones de los apartados a) hasta f)) para que el array dinámico esté definido con un **tipo de dato abstracto** mediante el siguiente `struct`. En este caso la función `inicializar` crea el array dinámico con una capacidad para 10 caracteres y la función `aniadir` aumentará la capacidad del array al doble de la actual, cuando al intentar añadir un nuevo carácter, la capacidad actual esté agotada. Este tipo es mejor que el anterior, ya que el array dinámico no debe ser redimensionado constantemente conforme se leen caracteres.

```
struct VectorDinamicoChar{
   char* arrayChar; // array dinámico de char
   int nCaracteres; // nº de caracteres actualmente en el array
   int capacidad;   // capacidad del array
}
```

(i) Haz una nueva versión de todo el programa (`main` y funciones de los apartados a) hasta f)) para que el array dinámico esté definido con un **tipo de dato abstracto** mediante la siguiente `clase`, análoga al `struct` del apartado anterior.

```
class VectorDinamicoChar{
   char* arrayChar; // array dinámico de char
   int nCaracteres; // nº de caracteres actualmente en el array
   int capacidad;   // capacidad del array
}
```

15. Queremos hacer un programa (`mostrar.cpp`) que lea los números enteros que tenemos almacenados en un fichero de texto pasado al programa como argumento de `main()` (o bien de la entrada estándar si no se pasa tal argumento) y que los cargue en un array dinámico `VectorSD`. El código de la función `main()` se da a continuación.

```
#include <iostream>
#include <fstream> // ifstream
#include "VectorSD.h"
using namespace std;

int main(int argc, char* argv[])
{
   VectorSD v;
   if (argc==1)
      v.leer(cin);
   else {
      ifstream flujo;
      flujo.open(argv[1]);
      if (!flujo) {
```

```
        cerr << "Error: Fichero " << argv[1] << " no válido." << endl;
        return 1;
      }
      v.leer(flujo);
      flujo.close();
   }
   v.mostrar(cout);
   v.liberar(); // Libera la memoria dinámica reservada
}
```

Como puede verse, el programa puede ser ejecutado de dos formas:

(a) Dando un argumento de entrada a main con el nombre del fichero de texto:

```
prompt> ./mostrar datos.txt
7 4 4 -3 13 16 16 -6 9 21
```

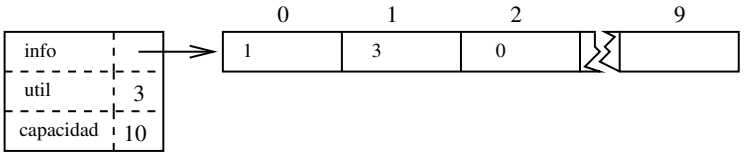
(b) Leyendo los datos de la entrada estándar:

```
prompt> cat datos.txt | ./mostrar
7 4 4 -3 13 16 16 -6 9 21
```

Construye el tipo de dato abstracto `VectorSD` con la siguiente clase:

```
class VectorSD {
   int *info; // array dinámico de int
   int util;  // nº de enteros actualmente en el array
   int capacidad;  // capacidad del array
};
```

donde `info` es un puntero que mantiene la dirección de una secuencia de enteros, `util` indica el número de componentes usados de la secuencia y `capacidad` indica el número de posiciones reservadas de la memoria dinámica para almacenar la secuencia de datos. La siguiente figura muestra un ejemplo de este tipo de representación.



Debemos construir este tipo de dato abstracto escribiendo los ficheros `VectorSD.h` y `VectorSD.cpp`.

Añadir a la clase los siguientes constructores y métodos:

(a) Constructor sin parámetros que inicialice una variable de tipo `VectorSD` reservando 10 casillas de memoria dinámica y ponga el número de componentes usadas a 0.

(b) Constructor de la clase que inicialice una variable de tipo `VectorSD` reservando **n** casillas de memoria dinámica y ponga el número de componentes usadas a 0.

(c) Método que nos devuelva el dato (`int`) almacenado en una determinada posición (`int`).

```
int getDato(int posicion) const {
   ...
}
```

(d) Método que devuelva el número de datos guardados actualmente en un objeto `VectorSD`.

```
int nElementos() const {
   ...
}
```

(e) Método que añada un nuevo dato a un objeto `VectorSD`. Considerar el caso de que la inclusión del nuevo valor sobrepase el número de elementos reservados. En este caso, realojar el array reservando el doble de posiciones.

```
void aniadir(int dato){
   ...
}
```

(f) Método que copie el objeto `VectorSD` recibido como parámetro en el objeto actual. La copia debe reservar memoria para almacenar solo las componentes usadas del array dinámico del objeto recibido por parámetro. Recuerde, que en caso de que el objeto ya contenga memoria dinámica reservada, debe liberarla antes.

```
void copia(const VectorSD &vector){
   ...
}
```

(g) Método que libere la memoria reservada por un objeto `VectorSD`.

```
void liberar(){
   ...
}
```

(h) Método para mostrar los elementos de un objeto `VectorSD` (separados por blancos) en un flujo de salida. Este método tendrá la siguiente forma:

```
void mostrar(ostream& flujo) const {
  for(int i=0; i<util;i++){
    flujo << info[i] << " ";
  }
  flujo<<endl;
}
```

(i) Método que recibe un flujo de entrada y carga uno a uno los datos enteros que contiene, en el objeto `VectorSD` hasta que llega al final de la entrada.

```
void leer(istream& flujo){
   int dato;

   while(flujo>>dato){
      aniadir(dato);
   }
}
```

16. Construye los tipos de datos abstractos Punto y Poligono mediante las clases `Punto` y `Poligono`. Para ello construye los correspondientes ficheros `.h` y `.cpp`. Un objeto `Punto` representa un punto en un espacio bidimensional. Un objeto `Poligono` permitirá almacenar polígonos convexos, con un número de vértices cualquiera. Para cada vértice, usaremos la clase `Punto`:

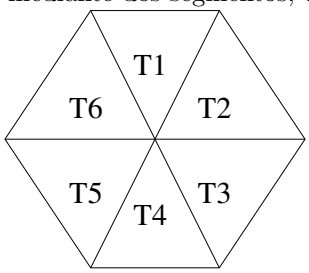
```
class Punto{
    double x;
    double y;
public:
    Punto(){x=0; y=0;};
    Punto(int x, int y){this->x=x; this->y=y;};
    double getX() const{return x;};
    double getY() const{return y;};
    void setXY(int x, int y){this->x=x; this->y=y;};
};
```

La clase `Poligono` guarda los vértices en un array dinámico de objetos `Punto` del tamaño justo para contener los vértices que tiene el polígono actualmente:

```
class Poligono{
    int nVertices;  // Número de puntos en el array dinámico
    Punto *vertices; // Array dinámico de objetos Punto
public:
    Poligono();
    Poligono(int n);
    void liberar();
    int getNumeroVertices() const;
    Punto getVertice(int index) const;
    void addVertice(const Punto& v);
    double getPerimetro() const;
};
```

(a) Implementa los métodos de las clases `Punto` y `Poligono` indicados en las declaraciones anteriores.

(b) Añade el método `double getArea()` a la clase `Poligono` para calcular el área de un polígono. Para ello, tenga en cuenta que el área de un polígono puede calcularse mediante la suma de las áreas de un conjunto de triángulos que están inscritos en él. Los triángulos se obtienen uniendo los lados del polígono con un punto interior, mediante dos segmentos, tal como muestra la siguiente figura.



Para calcular el área de un tríangulo, añade una función externa en el módulo `Poligono`: `double areaTriangulo(const Punto& pto1, const Punto& pto2, const Punto& pto3)`.
Se usará la siguiente fórmula para calcular el área:

$$area = \sqrt{T(T-S_1)(T-S_2)(T-S_3)}$$
$$T = \frac{S_1+S_2+S_3}{2}$$

donde S_1, S_2 y S_3 son las longitudes de los lados del triángulo.
Debe implementarse también en la clase `Poligono` el método auxiliar:
`Punto getPuntoInterior()`
que devuelve un punto interior cualquiera del polígono. Este método puede implementarse como el punto medio de la recta entre dos vértices opuestos o bien simplemente devolviendo uno de los vértices del polígono, ya que un vértice es también un punto interior al `Poligono`.

(c) Añade el método `void mostrar(ostream& flujo) const` para mostrar un objeto `Poligono` en un flujo de salida (muestra el número de vértices, y luego las coordenadas x e y de cada vértice). Por ejemplo, a continuación se muestra un polígono de 6 vértices:

```
6
4 12
10 12
14 6
10 0
4 0
0 6
```

(d) Añade el método `void leer(istream& flujo)` para leer un objeto `Poligono` de un flujo de entrada (en el formato del apartado anterior).

(e) Haz un programa (`leePoligono.cpp`) que lea un Polígono almacenado en un fichero de texto pasado al programa como argumento de `main()` (o de la entrada estándar si no se proporciona el argumento), lo muestre en la salida estándar y calcule su área y perímetro.

(f) Haz una nueva versión de la clase `Poligono`, añadiendo el dato miembro `int capacidad` para guardar la capacidad del array dinámico. Así, el array puede tener una capacidad mayor al número de vértices que tenga actualmente (`nVertices`), no necesitando ser redimensionado cada vez que se añade un vértice. Deben tenerse en cuenta los siguientes aspectos:

- El constructor sin parámetros creará el array de puntos con una capacidad inicial para `4` vértices. También inicializará a `0` el dato miembro `nVertices` para indicar que aun no contiene ningún punto.
- Un cliente (programa que use esta clase) de esta clase podrá añadir los vértices haciendo uso del método `void addVertice(const Punto& v)`.

El método `void addVertice(const Punto& v)` debe añadir el vértice en la primera posición libre del array dinámico al `Poligono` actual.

- Antes de añadir el vértice, este método debe comprobar si es posible añadirlo (o sea, si cabe en el array).

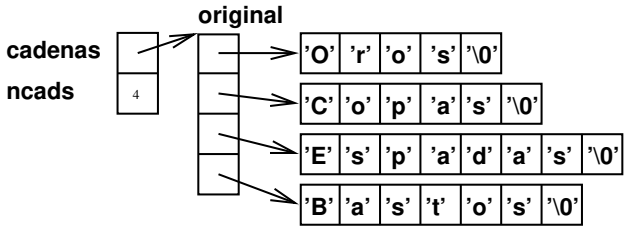
  Si no cabe, deberá crearse un nuevo array para los puntos con el doble de capacidad a la que tenga actualmente, y se copiarán los puntos antiguos en el nuevo array.
- El punto pasado como parámetro se añadirá en la primera posición disponible del array (o sea, en la posición `numVertices`).

4.2 Matrices dinámicas

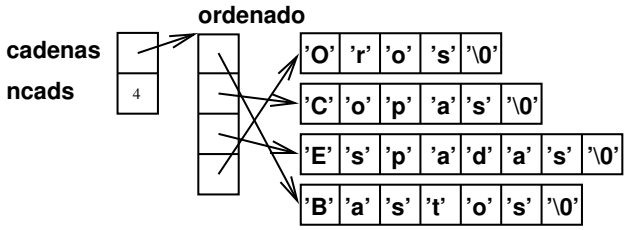
17. Queremos hacer un programa `ordenaLineas` que ordene por orden alfabético una serie de cadenas-C. Para ello construiremos un tipo de dato abstracto para el siguiente tipo definido con un `struct`:

```
struct VectorCadenas{
  char** cadenas; // array dinámico de cadenas tipo C
  int ncads; // Número de cadenas actualmente en el array
}
```

Este tipo representa un array dinámico de cadenas-C que almacenaremos en memoria dinámica. Por ejemplo, la siguiente figura muestra un ejemplo con cuatro cadenas:



La siguiente figura muestra la estructura tras ordenar el array de cadenas.



Debemos hacer un programa (`ordenarLineas.cpp`) que lea las cadenas de un fichero o de la entrada estándar, las vaya guardando en el array de cadenas y luego las ordene por orden alfabético. Guardaremos en memoria dinámica tanto el array de cadenas como cada una de las cadenas leídas. El array de cadenas tendrá en todo momento el tamaño justo necesario para guardar el número de cadenas que almacene. Cada cadena se almacena también en memoria dinámica con el tamaño justo necesario para guardar sus caracteres.

Para leer cada cadena, podemos hacer uso de la función `getline()` para así permitir espacios en blanco.

La función `main()` se da a continuación:

```
#include <iostream>
#include <fstream> // ifstream
#include "VectorCadenas.h"
using namespace std;

int main(int argc, char* argv[])
{
   VectorCadenas vCadenas;

   inicializar(vCadenas);
   if (argc==1)
      leer(cin, vCadenas);
```

```
   else {
      ifstream f(argv[1]);
      if (!f) {
         cerr << "Error: ordenarLineas " << argv[1] << " no válido." << endl;
         return 1;
      }
      leer(f, vCadenas);
   }
   cout << "Original:" << endl;
   mostrar(cout, vCadenas);
   ordenar(vCadenas);
   cout << "\nResultado:" << endl;
   mostrar(cout, vCadenas);
   liberar(vCadenas); // Libera la memoria dinámica reservada
}
```

Como puede verse, el programa puede ser ejecutado de dos formas:

(a) Dando un argumento de entrada a main con el nombre del fichero de texto:

```
prompt> ./ordenaLineas fichero.txt
Original:
Úbeda Martínez, Alejandro
Sabiote Alvarado, Lucía
Cano Granados, Maite

Resultado:
Cano Granados, Maite
Sabiote Alvarado, Lucía
Úbeda Martínez, Alejandro
```

(b) Leyendo los datos de la entrada estándar:

```
prompt> cat fichero.txt | ./ordenaLineas
Original:
Úbeda Martínez, Alejandro
Sabiote Alvarado, Lucía
Cano Granados, Maite

Resultado:
Cano Granados, Maite
Sabiote Alvarado, Lucía
Úbeda Martínez, Alejandro
```

Escribe los ficheros `VectorCadenas.h` y `VectorCadenas.cpp` del tipo de dato abstracto anterior. El módulo contendrá las siguientes funciones:

(a) La función `void inicializar(VectorCadenas& vectorCadenas)` que inicializa un array de cadenas como vacío.

(b) La función `void mostrar(ostream& flujo, const VectorCadenas& vectorCadenas)` para mostrar en un flujo de salida las cadenas almacenadas en un array de cadenas (`VectorCadenas`).

(c) La función `void redimensionar(VectorCadenas& vectorCadenas,int newSize)`.

(d) La función `void aniadir(VectorCadenas& vectorCadenas, const char* cadena)`.

(e) La función `void leer(istream& flujo, VectorCadenas& vectorCadenas)` que lee todas las líneas que contenga un flujo de entrada y las guarde en el objeto recibido como parámetro. Esta función se da a continuación:

```
void leer(istream& flujo, VectorCadenas& vectorCadenas){
   const int NCARACTERES=1000; // Suponemos líneas con menos de 1000 char
   char linea[NCARACTERES];
```

```
   while(flujo.getline(linea,NCARACTERES)){
      aniadir(vectorCadenas, linea);
   }
}
```

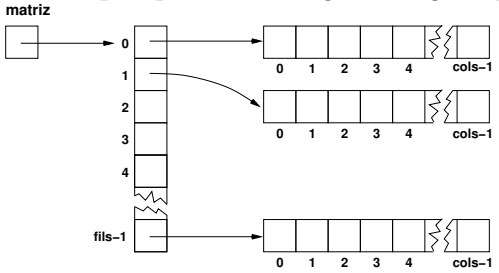
(f) La función `void liberar(VectorCadenas& vectorCadenas)` libera la memoria dinámica del objeto `vectorCadenas`.

(g) La función `void ordenar(VectorCadenas& vectorCadenas)` para ordenar las cadenas por orden alfabético.

(h) Haz una nueva versión de todo el programa (`main` y funciones del tipo de dato abstracto) para que el vector dinámico de cadenas de caracteres, esté definido con un **tipo de dato abstracto** mediante la siguiente `clase`.

```
class VectorCadenas{
  char** cadenas; // array dinámico de cadenas tipo C
  int ncads; // Número de cadenas actualmente en el array
  int capacidad: // Capacidad del array dinámico
}
```

18. Queremos construir un tipo de dato abstracto para almacenar matrices bidimensionales dinámicas. Para ello usamos una estructura como la que aparece en la siguiente figura que denominaremos **Matriz2D_1**:



(a) Define cómo sería la clase (fichero `.h`) para representar una matriz según esa estructura.

(b) Añade el constructor de la clase, para crear una matriz con `nf` filas y `nc` columnas.

(c) Métodos para obtener el número de filas de la matriz y el número de columnas.

(d) Métodos para obtener y modificar el dato que hay en una determinada fila y columna.

(e) Método que libere la memoria dinámica ocupada por un objeto de la clase.

(f) Método que escriba en un flujo de salida el número de filas y de columnas de la matriz, y a continuación todos sus componentes. Por ejemplo, una matriz de 3 filas y 4 columnas se mostraría de la siguiente forma:

```
3 4
4 12 10 12
14 6 10 0
4 0 0 6
```

(g) Método que lea datos de un flujo de entrada (con el formato del apartado anterior) y los guarde en la matriz dinámica

(h) Método que dada una matriz de este tipo cree una copia.

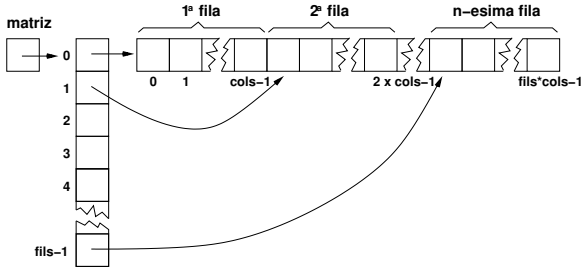
(i) Método que extraiga una submatriz de una matriz bidimensional `Matriz2D_1`. Como argumento del método se introduce desde qué fila y columna y hasta qué fila y columna se debe realizar la copia de la matriz original.

(j) Método que elimine una fila de una matriz bidimensional `Matriz2D_1`. Obviamente, la eliminación de una fila implica el desplazamiento hacia arriba del resto de las filas que se encuentran por debajo de la fila eliminada.

(k) Método como el anterior, pero que en vez de eliminar una fila, elimine una columna.

(l) Escribe un programa que lea datos de un flujo de entrada y los guarde en una matriz dinámica. Una vez leída la matriz, el programa calculará la **matriz traspuesta** y la enviará a un flujo de salida según el mismo formato.

19. Supongamos que ahora decidimos utilizar una forma diferente para representar las matrices bidimensionales dinámicas a la que se propone en el ejercicio anterior. En este caso, usaremos una estructura semejante a la que aparece en la siguiente figura que denominaremos **Matriz2D_2**:



(a) Realizar los apartados a) hasta k) usando esta nueva representación de matrices bidimensionales dinámicas.

(b) Construir un método que dada una matriz bidimensional dinámica `Matriz2D_1` realice una copia de la misma en una matriz bidimensional dinámica `Matriz2D_2`.

(c) Desarrollar un método que realice el paso inverso al propuesto en el apartado anterior.

(d) Escribe un programa que lea datos de un flujo de entrada y los guarde en una matriz dinámica. Una vez leída la matriz, el programa calculará la **matriz traspuesta** y la enviará a un flujo de salida según el mismo formato.

4.3 Listas de celdas enlazadas

20. Dada la siguiente definición de tipo de dato abstracto para definir una lista de celdas enlazadas:

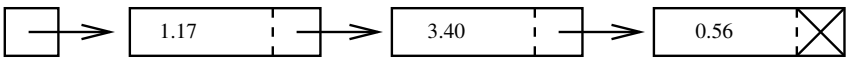
```
struct Celda{
   double info;
   Celda *sig;
};
```

donde `info` es una variable `double` y `sig` es un puntero que apunta a una variable `Celda`. Implementad las siguientes funciones:

a) Función que inicialice una variable de tipo `Lista`, como una lista vacía.

b) Función que permita añadir al final de una secuencia de celdas enlazadas un nuevo dato. Por ejemplo, dada la siguiente situación:

el resultado de añadir el valor `0.56` a la lista correspondiente sería:



NOTA: El puntero de la última celda contiene `nullptr`.

c) Función que permita eliminar la última celda de una lista.

d) Función que elimine y libere toda la información contenida en una lista.

e) Función que muestre en un flujo de salida el contenido completo de una lista.

f) Función que inserte un nuevo dato detrás de una celda concreta (cuya dirección de memoria se pasa como parámetro al método).

g) ¿Podría utilizarse la función anterior para insertar una celda al principio de la lista? Si no es así, haced los cambios necesarios para que sea posible.

h) Función que, dada la dirección de memoria de una celda, la elimine de la lista.

i) ¿Sería posible utilizar la función anterior para eliminar la primera de las celdas de la estructura de celdas enlazadas? Si no es así, haced los cambios necesarios para que sea posible.

j) Función que devuelva un puntero a la celda que se encuentra en una determinada posición (entera) en la lista.

k) Hacer un programa (mostrar.cpp) que lea los números reales que tenemos almacenados en un fichero de texto pasado al programa como argumento de main() (o bien de la entrada estándar si no se pasa tal argumento) y que los cargue en una lista de celdas enlazadas. Finalmente, el programa debe mostrar el contenido de la lista desde el principio hasta el final.

21. Dada la siguiente definición de tipo de dato abstracto para definir una lista de celdas enlazadas:

```
struct Celda{
   double info;
   Celda *sig;
};

class Lista{
   Celda *l;
public:
   ....
};
```

donde `info` es una variable `double` y `sig` es un puntero que apunta a una variable `Celda`. Construye la clase `Lista` con los siguientes métodos y constructores:

- Constructor sin parámetros para crear una lista vacía.
- `print(ostream& flujo)`: Muestra el contenido de la lista en un flujo de salida.
- `add(int index, double element)`: Inserta un elemento en la posición especificada (`index`).
- `addFirst(double element)`: Inserta un elemento al principio de la lista.
- `addLast(double element)`: Inserta un elemento al final de la lista.
- `clear()`: Borrar todos los elementos de la lista.
- `copy()`: Devuelve una copia de la lista.
- `contains(double element)`: Devuelve si la lista contiene el elemento.
- `get(int index)`: Devuelve el elemento que se encuentra en la posición indicada.
- `getFirst()`: Devuelve el primer elemento de la lista.
- `getLast()`: Devuelve el último elemento de la lista.
- `indexOf(double element)`: Devuelve el índice de la primera ocurrencia del elemento especificado en esta lista, o -1 si la lista no contiene el elemento.
- `lastIndexOf(double element)`: Devuelve el índice de la última ocurrencia del elemento especificado en esta lista, o -1 si la lista no contiene el elemento.
- `remove(int index)`: Borra el elemento que está en la posición indicada.
- `removeFirst()`: Borra el primer elemento de la lista.
- `removeLast()`: Borra el primer elemento de la lista.
- `set(int index, double element)`: Modifica el elemento de la posición indicada de esta lista.
- `size()`: Devuelve el número de elementos de esta lista.

Finalmente, hacer un programa (mostrar.cpp) que lea los números reales que tenemos almacenados en un fichero de texto pasado al programa como argumento de main() (o bien de la entrada estándar si no se pasa tal argumento) y que los cargue en una lista de celdas enlazadas. Finalmente, el programa debe mostrar el contenido de la lista desde el principio hasta el final.

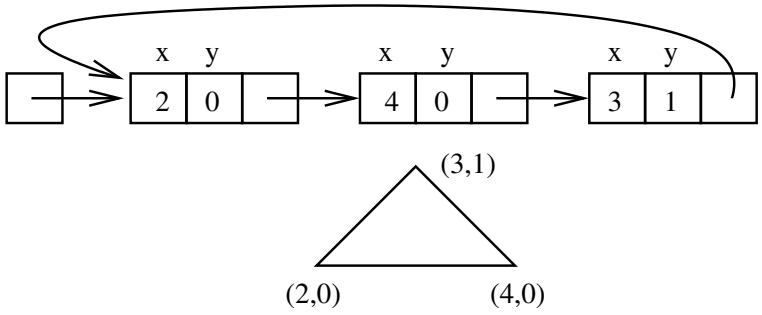
22. Se desea desarrollar una clase que permita representar de forma general diversas figuras poligonales. Cada figura poligonal se puede representar como un conjunto de puntos en el plano unidos por segmentos de rectas entre cada dos puntos adyacentes. Por esta razón se propone la siguiente representación:

```
struct Punto2D{
   double x;
   double y;
};

struct Nodo{
   Punto2D punto;
   Nodo *sigPunto;
};

class Poligono{
  Nodo* poligono;
}
```

Dada esta definición, un polígono se representa como una secuencia circular ordenada de nodos enlazados, por ejemplo, el triángulo de puntos (2,0),(4,0) y (3,1) se representa de la siguiente forma:



Teniendo en cuenta está representación, responder a las siguientes cuestiones:

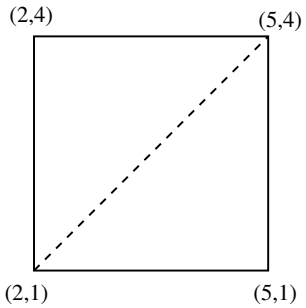
a) Construir un método en la clase `Poligono` que determine el número de lados que contiene la figura almacenada en una variable de tipo `Poligono`.

b) Suponiendo que existe un método llamado `pintaRecta (const Punto2D &p1, const Punto2D &p2)` que pinta una recta entre los 2 puntos que se le pasan como argumento, construir un método que permita pintar la figura que representa una determinada variable `Poligono`.

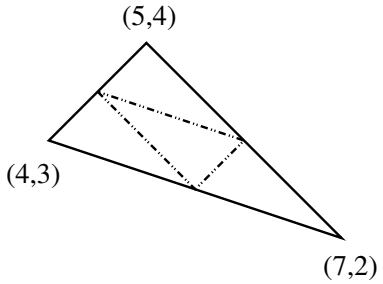
c) Implementar un método que permita crear en una variable de tipo `Poligono`, un triángulo a partir de los tres puntos que lo definen.

d) Desarrollar un método que permita liberar la memoria reservada por una variable `Poligono`.

e) Sabiendo que una variable `Poligono` almacena un cuadrado, implementar un método que devuelva los dos triángulos que resultan de unir mediante una recta la esquina inferior izquierda del cuadrado con su esquina superior derecha.



f) Construir un método que a partir de una variable `Poligono` que representa un triángulo devuelva el triángulo formado por los puntos medios de las rectas del triángulo original.



g) Desarrollar un constructor que permita construir un polígono regular de **n** lados inscrito en una circunferencia de radio **r** y centro **(x,y)**.