Programación y Estructuras de Datos

CUADERNILLO 1

(curso 2009-2010, 1er. Cuatrimestre)

Introducción **: Contexto del problema**

En la industria de la transformación del mármol, existe un proceso que es el de rellenado de grietas, poros y fisuras.

Una plancha de mármol cuando se extrae de la cantera tiene grietas y poros que se tienen que tapar con un producto similar al mármol en cuanto a dureza y solidez. Este producto se llama *apoxi*. El *apoxi* es un producto caro, incoloro (se le pueden añadir colorantes para obtener *apoxi* con un color concreto) y muy difícil de dosificar, por lo que en la actualidad este producto se aplica manualmente sobre el mármol extendiéndolo con una paleta, proceso en el cual se invierte mucho tiempo.

Con el fin de ahorrar tiempo (y reducir el número de empleados), pretendemos crear un sistema que dosifique este producto sobre las grietas y fisuras del mármol de manera automática.

En el mercado existen unas pistolas de dosificación preparadas para el *apoxi*. Sin embargo, estas pistolas tienen un funcionamiento muy peculiar: la pistola de dosificación está dotada de dos válvulas de salida. Una está conectada a la cámara principal y la otra está conectada a la cámara secundaria.

La plancha de mármol a la que tenemos que tapar las grietas tiene poros de distintas dimensiones (desde muy grandes hasta muy pequeños). La secuencia óptima es tapar con la cámara principal el poro máximo y con la cámara secundaria el poro mínimo.

Por tanto, se pretende crear una serie de estructuras de datos que nos permitan acceder de una forma óptima a los poros que tienen la mayor y la menor dimensión de todos.

Parte 1 **: clase base "TPoro"**

Qué se pide :

Se pide construir una clase que representa un PORO EN UNA PLANCHA DE MÁRMOL (su posición, su volumen y su color).

Cuándo se exige:

Al alumno se le podrá exigir tenerlo terminado y funcionando a partir del día : **Lunes 9/11/2009**

Prototipo de la Clase:

PARTE PRIVADA

```
// Coordenada x de la posición
int x;

// Coordenada y de la posición
int y;

// Volumen
double volumen;

// Color
char* color;
```

FORMA CANÓNICA

```
// Constructor por defecto
TPoro();

// Constructor a partir de una posición y un volumen
TPoro(int, int, double);

// Constructor a partir de una posición, un volumen y un color
TPoro(int, int, double, char *);

// Constructor de copia
TPoro(TPoro &);

// Destructor
~TPoro();

// Sobrecarga del operador asignación
TPoro & operator=(TPoro &);
```

MÉTODOS

```
// Sobrecarga del operador igualdad
bool operator==(TPoro &);

// Sobrecarga del operador desigualdad
bool operator!=(TPoro &);

// Modifica la posición
void Posicion(int, int);

// Modifica el volumen
void Volumen(double);

// Modifica el color
void Color(char *);

// Devuelve la coordenada x de la posición
int PosicionX();

// Devuelve la coordenada y de la posición
int PosicionY();

// Devuelve el volumen
double Volumen();

// Devuelve el color
char * Color();

// Devuelve cierto si el poro está vacío
bool EsVacio();
```

FUNCIONES AMIGAS

```
// Sobrecarga del operador SALIDA
friend ostream & operator<<(ostream &, TPoro &);
```

Aclaraciones :

- La posición por defecto es (0, 0), el volumen por defecto es 0 y el color por defecto es una cadena vacía (puntero a NULL) : sólo cumpliendo TODAS estas condiciones, se considerará que el elemento **TPoro** es "vacío".

- Asimismo, las anteriores características son las que definirán a un **TPoro** inicializado con el **Constructor por defecto.**

- Los colores se tienen que escribir en minúsculas: si en algún momento se emplea un color en mayúsculas (por ejemplo, en el constructor), cuando se almacene se tiene que pasar a minúsculas.

- El **destructor** tiene que liberar toda la memoria que ocupe el objeto, asignando todos los valores numéricos a 0.

- En el operador **igualdad**, dos poros son iguales si poseen la misma posición, el mismo volumen y el mismo color.

- El **operador salida** muestra el contenido del **TPoro** .

  Primero se muestra la posición entre paréntesis, separando con una coma la coordenada X, que debe ir primero, de la coordenada Y .
  A continuación separado por un espacio en blanco el volumen (con 2 decimales) y por último, también separado por un espacio en blanco, el color.
  Si el color es un puntero a NULL, se tiene que mostrar un guión (el signo menos).
  Si el objeto está vacío, se mostrará la cadena "()".
  NO se tiene que generar un salto de línea al final.

  Ejemplo:

```
(4, 5) 2.31 azul

(0, 10) 12.15 -

()
```

Parte 2 **: vector de TPoro "TVectorPoro"**

Qué se pide :

Se pide construir una clase que representa un VECTOR DE ELEMENTOS de la clase base **TPoro.**

Cuándo se exige:

Al alumno se le podrá exigir tenerlo terminado y funcionando a partir del día : **Lunes 23/11/2009**

Prototipo de la Clase:

PARTE PRIVADA

```
// Dimension del vector
int dimension;

// Datos del vector
TPoro *datos;

// Para cuando haya error en el operator[]
TPoro error;
```

FORMA CANÓNICA

```
// Constructor por defecto
TVectorPoro();

// Constructor a partir de una dimensión
TVectorPoro(int);

// Constructor de copia
TVectorPoro(TVectorPoro &);

// Destructor
~TVectorPoro();

// Sobrecarga del operador asignación
TVectorPoro & operator=(TVectorPoro &);
```

MÉTODOS

```
// Sobrecarga del operador igualdad
bool operator==(TVectorPoro &);

// Sobrecarga del operador desigualdad
bool operator!=(TVectorPoro &);

// Sobrecarga del operador corchete (parte IZQUIERDA)
TPoro & operator[](int);

// Sobrecarga del operador corchete (parte DERECHA)
TPoro operator[](int) const;


// Devuelve la longitud (dimensión) del vector
int Longitud();

// Devuelve la cantidad de posiciones ocupadas (no vacías) en el vector
int Cantidad();

// REDIMENSIONAR el  vector de TPoro
bool  Redimensionar(int);
```

FUNCIONES AMIGAS

```
// Sobrecarga del operador salida
friend ostream & operator<<(ostream &, TVectorPoro &);
```

Aclaraciones :

- El vector NO tiene por qué estar ordenado.
- El vector PUEDE contener elementos repetidos. Incluso de los considerados elementos **TPoro** "vacíos"**.**
- Se consideran elementos **TPoro** "vacíos", aquellos que contienen la posición igual a (0, 0), el volumen igual a 0 y el color igual a una cadena vacía (puntero a NULL)**.**
- El **constructor por defecto** crea un vector de dimensión 0 (puntero interno a NULL, no se reserva memoria).
- En el **constructor a partir de una dimensión**, si la dimensión es menor o igual que 0, se creará un vector de dimensión 0 (como el constructor por defecto).
- Si se asigna un vector a un vector no vacío, se destruye el vector inicial. Puede ocurrir que se modifique la dimensión del vector al asignar un vector más pequeño o más grande.
- El **destructor** tiene que liberar toda la memoria que ocupe el vector, quedando la dimensión igual a 0.
- En el **operador igualdad**, dos vectores son iguales si poseen la misma dimensión y los mismos elementos en las mismas posiciones.
- En la sobrecarga del **operador corchete**, las posiciones van desde 1 hasta la ***dimensión*** del vector.

  Si se accede a una posición que no existe, se tiene que devolver un objeto **TPoro** "vacío". Para ello se declara en la parte privada un elemento de clase de nombre "**error**" (para devolverlo por referencia)

- En el **operador "Redimensionar" :**

  A esta función se le pasa un entero y hay que redimensionar el vector al tamaño del entero.

  La salida será TRUE cuando se haya producido realmente un redimensionamiento, y FALSE cuando el vector permanezca con la dimensión antigua.

  Los valores del entero que se pasa por parámetro pueden ajustarse a estos casos :

  - ✓ Si el entero es menor o igual que 0 , el método devolverá FALSE, sin hacer nada más.
  - ✓ Si el entero es de igual tamaño que el actual del vector, el método devolverá FALSE, sin hacer nada más.
  - ✓ Si el entero es mayor que 0 y mayor que el tamaño actual del vector, hay que copiar los componentes del vector en el vector nuevo, que pasará a tener el tamaño que indica el entero. Las nuevas posiciones serán vacías, es decir, objetos **TPoro** inicializados con el **Constructor por defecto** de **TPoro**.
  - ✓ Si el entero es mayor que 0 y menor que el tamaño actual del vector, se deben eliminar los **TPoro** que sobren por la derecha, dejando el nuevo tamaño igual al valor del entero.

- El **operador salida** muestra el contenido del vector desde la primera hasta la última posición en una sola línea. Todo el contenido del vector se muestra entre corchetes "[ ]".
  Para cada elemento, primero se mostrará la posición del elemento y a continuación, separado por un espacio en blanco, el elemento en sí.
  Cada elemento se tiene que separar del siguiente por un espacio en blanco (a continuación del último elemento no se tiene que mostrar nada).
  NO se tiene que generar un salto de línea al final. Si la dimensión del vector es 0, se tiene que mostrar la cadena "[]".

  Por ejemplo:

```
[]

[1 (4, 5) 2.31 azul 2 (0, 10) 12.15 - 3 ()]
```

Parte 3 **: pila de TPoro "TPilaPoro"**

Qué se pide :

Se pide construir una clase que representa una PILA de elementos de la clase base **TPoro,** a partir de la estructura de datos **TVectorPoro** definida en la **Parte 2.**

Cuándo se exige:

Al alumno se le podrá exigir tenerlo terminado y funcionando a partir del día : **Lunes 30/11/2009**

Prototipo de la Clase:

PARTE PRIVADA

```
// Almacena los poros
TVectorPoro v;

// Indica la próxima posición libre en el vector
int posicion;

// Para cuando haya que devolver un TPoro "vacío"
TPoro error;
```

FORMA CANÓNICA

```
// Constructor por defecto
TPilaPoro();

// Constructor de copia
TPilaPoro(TPilaPoro &);

// Destructor
~TPilaPoro();

// Sobrecarga del operador asignación
TPilaPoro & operator=(TPilaPoro &);
```

MÉTODOS

```
// Sobrecarga del operador igualdad
bool operator==(TPilaPoro &);

// Devuelve TRUE si la pila está vacía, FALSE en caso contrario
bool EsVacia();

// Apila un poro en la pila
void Apilar(TPoro &);
```

```
// Desapila un poro de la cima de la pila
void Desapilar();

// Devuelve el poro situado en la cima de la pila
TPoro Cima();

// Devuelve la longitud de la pila
int Longitud();

// SUMA de 2 pilas
TPilaPoro operator+ (TPilaPoro &);
```

FUNCIONES AMIGAS

```
// Sobrecarga del operador salida
friend ostream & operator<<(ostream &, TPilaPoro &);
```

Aclaraciones :

- La pila define internamente por *composición* un objeto del tipo **TVectorPoro**.
- La pila puede contener elementos **TPoro** repetidos, incluso de los considerados "vacíos".
- El **constructor de copia** tiene que realizar una copia exacta (el vector que se crea ha de tener la misma dimensión que el vector copiado).
- El **constructor por defecto** crea una pila vacía que internamente posee un objeto **TVectorPoro** de dimensión 10.
- El **destructor** tiene que liberar toda la memoria que ocupe la pila.
- En el **operador asignación**, si se asigna una pila a una pila no vacía, se destruye la pila inicial. La asignación tiene que realizar una copia exacta (el vector que se crea ha de tener la misma dimensión que el vector copiado).
- En el **operador igualdad**, dos pilas son iguales si poseen los mismos elementos en el mismo orden (los vectores internos pueden poseer distintas dimensiones).
- Si la pila está vacía, **Cima**() devuelve un objeto **TPoro** "vacío" .
- **Longitud**() devuelve el número de elementos **TPoro** no "vacíos" que hay en la pila.
- **Apilar**: si el vector está lleno (todas las posiciones ocupadas), se tiene que crear un nuevo vector con 10 posiciones más, copiar el contenido del vector inicial al nuevo vector, insertar el nuevo elemento y eliminar el vector inicial.

  Devuelve *TRUE* si el elemento se puede apilar y *FALSE* en caso contrario (por ejemplo, porque no se pueda reservar memoria).

- **Desapilar** : si hay elementos no "vacíos" en la pila y antes de eliminar el elemento quedan 10 posiciones "vacías" en el vector, se tiene que crear un nuevo vector con 10 posiciones menos, copiar el contenido del vector inicial al nuevo vector, eliminar el elemento y eliminar el vector inicial.

Devuelve *TRUE* si el elemento se puede desapilar y *FALSE* en caso contrario (por ejemplo, porque la pila está vacía).

- El tamaño mínimo del vector es 10: si se desapilan todos los elementos, NO se tiene que crear un vector de dimensión 0. Es decir, el vector interno que almacena los datos crece o decrece de 10 en 10 posiciones según haga falta.

- El **operador suma** devuelve una pila con los elementos **TPoro** de la pila (operando izquierdo, objeto THIS), en el mismo orden en que se encontraban inicialmente en el operando izquierdo, más los elementos de la pila que actúa como operando derecho.

- El **operador salida** muestra el contenido de la pila desde la cima hasta la base de la pila en una sola línea.
  Todo el contenido de la pila se muestra entre llaves "{ }".
  Entre la llave de apertura y el primer elemento y entre el último elemento y la llave de cierre NO tienen que aparecer espacios en blanco.
  Cada elemento se tiene que separar del siguiente por un espacio en blanco (a continuación del último elemento NO se tiene que generar un espacio en blanco).
  NO se tiene que generar un salto de línea al final. Si la pila está vacía, se tiene que mostrar la cadena "{}".

Por ejemplo:

```
{}

{(4, 5) 2.31 azul (0, 10) 12.15 - ()}
```

Otro ejemplo (**suma** de pilas):

```
TPoro p1(1,1,1);
TPilaPoro pi1, pi2;
pi1.Apilar(p1);
pi2.Apilar(p1);
cout << p1 + p2 ;
```

(La salida sería) :

```
{(1, 1) 1 - (1, 1) 1 -}
```

Parte 4 **: lista de TPoro "TListaPoro"**

Qué se pide :

Se pide construir una clase que representa una LISTA ORDENADA Y DOBLEMENTE ENLAZADA de elementos de la clase base **TPoro.**

La lista estará ordenada según el volumen del poro (de menor a mayor)

Se trata de una lista doblemente enlazada (para poder recorrerla en ambos sentidos)

Será una lista de poros accesible por una posición. Existe una clase auxiliar para representar la posición (**TListaPosicion**).

Para representar cada NODO de la lista, se tiene que definir la clase **TListaNodo** con su forma canónica (**constructor, constructor de copia, destructor** y sobrecarga del **operador asignación**) como mínimo.

Cuándo se exige:

Al alumno se le podrá exigir tenerlo terminado y funcionando a partir del día : **Lunes 14/12/2009**

Prototipo de la Clase "**TListaNodo**":

// PARTE PRIVADA

```
// El elemento del nodo
TPoro e;

// El nodo anterior
TListaNodo *anterior;

// El nodo siguiente
TListaNodo *siguiente;
```

// FORMA CANÓNICA

```
// Constructor por defecto
TListaNodo ();

// Constructor de copia
TListaNodo (TListaNodo &);

// Destructor
~TListaNodo ();

// Sobrecarga del operador asignación
TListaNodo & operator=( TListaNodo &);
```

Prototipo de la Clase "**TListaPosicion**":

```
// PARTE PRIVADA

// Puntero a un nodo de la lista
TListaNodo *pos;


// FORMA CANÓNICA

// Constructor por defecto
TListaPosicion ();

// Constructor de copia
TListaPosicion (TListaPosicion &);

// Destructor
~TListaPosicion ();

// Sobrecarga del operador asignación
TListaPosicion& operator=( TListaPosicion &);


// MÉTODOS

// Sobrecarga del operador igualdad
bool operator==( TListaPosicion &);

// Devuelve la posición anterior
TListaPosicion Anterior();

// Devuelve la posición siguiente
TListaPosicion Siguiente();

// Devuelve TRUE si la posición no apunta a una lista, FALSE en caso contrario
bool EsVacia();
```

Prototipo de la Clase "TListaPoro":

PARTE PRIVADA

```
// Primer elemento de la lista
TListaNodo *primero;

// Ultimo elemento de la lista
TListaNodo *ultimo;
```

FORMA CANÓNICA

```
// Constructor por defecto
TListaPoro();

// Constructor de copia
TListaPoro (TListaPoro &);

// Destructor
~TListaPoro ();

// Sobrecarga del operador asignación
TListaPoro & operator=( TListaPoro &);
```

MÉTODOS

```
// Sobrecarga del operador igualdad
bool operator==(TListaPoro &);

// Sobrecarga del operador suma
TListaPoro operator+(TListaPoro &);

// Sobrecarga del operador resta
TListaPoro operator-(TListaPoro &);

// Devuelve true si la lista está vacía, false en caso contrario
bool EsVacia();

// Inserta el elemento en la lista
bool Insertar(TPoro &);

// Busca y borra el elemento
bool Borrar(TPoro &);

// Borra el elemento que ocupa la posición indicada
bool Borrar(TListaPosicion &);

// Obtiene el elemento que ocupa la posición indicada
TPoro Obtener(TListaPosicion &);

// Devuelve true si el elemento está en la lista, false en caso contrario
bool Buscar(TPoro &);
```

```
// Devuelve la longitud de la lista
int Longitud();

// Devuelve la primera posición en la lista
TListaPosicion Primera();

// Devuelve la última posición en la lista
TListaPosicion Ultima();
```

FUNCIONES AMIGAS

```
// Sobrecarga del operador salida
friend ostream & operator<<(ostream &, TListaPoro &);
```

Aclaraciones de la Clase "**TListaPosicion**" :

- Evidentemente, una posición puede dejar de ser válida en cualquier momento (por ejemplo, la lista a la que apunta la posición puede variar o incluso ser destruida). Este problema NO se ha de tener en cuenta en la realización de la práctica.
- En **"Anterior()"** y **"Siguiente()"**, si la posición actual es la primera o la última de la lista, se tiene que devolver una posición vacía.
- En " **EsVacia()**" : devuelve TRUE si el puntero interno ("pos") es NULL . En caso contrario devuelve FALSE.
- En el **operador igualdad**, dos posiciones son iguales si apuntan a la misma posición de la lista.

Aclaraciones de la Clase "**TListaPoro**" :

- La lista NO puede contener elementos repetidos. Si contiene un elemento **TPoro** "vacío", NO podrá haber otro igual.
- La lista está ordenada de menor a mayor, de acuerdo al VOLUMEN del objeto **TPoro** que contiene el nodo.
- El **constructor por defecto** crea una lista vacía.
- El **constructor de copia** tiene que realizar una copia exacta.
- El **destructor** tiene que liberar toda la memoria que ocupe la lista.
- **operador asignación**, si se asigna una lista a una lista no vacía, se destruye la lista inicial. La asignación tiene que realizar una copia exacta.
- En el **operador igualdad**, dos listas son iguales si poseen los mismos elementos en el mismo orden.
- El **operador suma** une los elementos de dos listas en una nueva lista (ordenada y sin repetidos).
- El **operador resta** devuelve una lista nueva que contiene los elementos de la primera lista (operando de la izquierda) que NO existen en la segunda lista (operando de la derecha).
- En **Insertar**, el nuevo elemento se inserta en la posición adecuada para que siga siendo una lista ordenada.

En caso de que el elemento a insertar contenga un VOLUMEN igual a uno ya existente en la lista, el nuevo nodo se insertará DESPUÉS (en orden), al que ya existía.
Devuelve TRUE si el elemento se puede insertar y FALSE en caso contrario (por ejemplo, porque ya exista en la lista).

- En **Borrar**, devuelve TRUE si el elemento se puede borrar y FALSE en caso contrario (por ejemplo, porque el elemento no existe en la lista).

- En **Borrar**, el paso por referencia es obligatorio, ya que una vez eliminado el elemento la posición tiene que pasar a esta vacía (no asignada a ninguna lista).

  Además, devuelve TRUE si el elemento se puede borrar y FALSE en caso contrario (por ejemplo, porque la posición no pertenece a la lista).

- En **Obtener**, si la posición no pertenece a la lista se tiene que devolver un objeto **TPoro** "vacío".

- En **Buscar**, NO hace falta implementar una búsqueda avanzada (por ejemplo, una búsqueda binaria). Se puede realizar una búsqueda lineal desde el primer elemento hasta el último.

- **Longitud** devuelve el número de elementos que hay en la lista ("vacíos" o "no vacíos").

- En **Primera** y **Ultima**, si la lista está vacía se tiene que devolver una posición vacía.

- El **operador salida** muestra el contenido de la lista desde la cabeza hasta el final de la lista.

  Todo el contenido de la lista se muestra entre paréntesis "( )".
  Entre el paréntesis de apertura y el primer elemento y entre el último elemento y el paréntesis de cierre NO tienen que aparecer espacios en blanco.
  Cada elemento se tiene que separar del siguiente por un espacio en blanco (a continuación del último elemento no se tiene que generar un espacio en blanco).
  NO se tiene que generar un salto de línea al final.
  Si la lista esta vacía, se tiene que mostrar la cadena "()"

Por ejemplo:

```
TPoro p1(1,1,1);
TListaPoro l1;
l1.Insertar (p1);
cout << l1 ;
```

(La salida sería) :

```
((1, 1) 1 -)
```

ANEXO 1. Notas de aplicación general sobre el contenido del Cuadernillo.

Cualquier modificación o comentario del enunciado se publicará oportunamente en el Campus Virtual.

En su momento, se publicarán como materiales del Campus Virtual los distintos FICHEROS de MAIN (**tad.cpp**) que servirán al alumno para realizar unas pruebas básicas con los TAD propuestos.
No obstante, se recomienda al alumno que aporte sus propios ficheros **tad.cpp** y realice sus propias pruebas con ellos.
(El alumno tiene que crearse su propio conjunto de ficheros de prueba para verificar de forma exhaustiva el correcto funcionamiento de la práctica. Los ficheros que se publicarán en el CV son sólo una muestra y en ningún caso contemplan todos los posibles casos que se deben verificar)

Todas las operaciones especificadas en el Cuadernillo son obligatorias.

Si una clase hace uso de otra clase, en el código nunca se debe incluir el fichero **.cpp**, sólo el **.h**.

El paso de parámetros como constantes o por referencia se puede cambiar dependiendo de la representación de cada tipo y de los algoritmos desarrollados. Del mismo modo, el alumno debe decidir si usa el modificador CONST , o no.

En la parte PUBLIC no debe aparecer ninguna operación que haga referencia a la representación del tipo, sólo se pueden añadir operaciones de enriquecimiento de la clase.

En la parte PRIVATE de las clases se pueden añadir todos los atributos y métodos que sean necesarios para la implementación de los tipos.

Tratamiento de **excepciones**: todos los métodos darán un mensaje de error (en **cerr**) cuando el alumno determine que se produzcan **excepciones**; para ello, se pueden añadir en la parte privada de la clase aquellas operaciones y variables auxiliares que se necesiten para controlar las excepciones.

Se considera **excepción** aquello que no permite la normal ejecución de un programa (por ejemplo, problemas al reservar memoria, problemas al abrir un fichero, etc.). **NO** se considera **excepción** aquellos errores de tipo **lógico** debidos a las especificidades de cada clase .

De cualquier modo, todos los métodos deben devolver siempre una variable del tipo que se espera.
Los mensajes de error se mostrarán siempre por la salida de error estándar (cerr). El formato será:

ERROR: mensaje_de_error *(al final un salto de línea).*

ANEXO 2. Condiciones de ENTREGA.

2.1. Dónde, cómo, cuándo.

La entrega de la práctica se realizará :

- Servidor: en el SERVIDOR DE PRÁCTICAS, cuya URL es : http://pracdlsi.dlsi.ua.es/
- Fecha: las fechas de entrega y Examen Práctico se publicarán en Campus Virtual oportunamente.
- A título INDIVIDUAL (aunque el alumno la haya realizado en pareja); por tanto requerirá del alumno que conozca su USUARIO y CONTRASEÑA en el Servidor de Prácticas.

2.2. Ficheros a entregar y comprobaciones.

La práctica debe ir organizada en 3 subdirectorios:

DIRECTORIO 'include' : contiene los ficheros (nombres en MINÚSCULAS) :
- **"tporo.h"**
- **"tvectorporo.h"**
- **"tpilaporo.h"**
- **"tlistaporo.h" (**incluye clases **: TListaPoro , TListaNodo y TListaPosicion)**

DIRECTORIO 'lib' : contiene los ficheros **:**
- **" tporo.cpp"**
- **" tvectorporo.cpp"**
- **" tpilaporo.cpp"**
- **" tlistaporo.cpp" (**incluye clases **: TListaPoro, TListaNodo y TListaPosicion)**

DIRECTORIO 'src' : contiene los ficheros :
- **"tad.cpp" (**fichero aportado por el alumo para comprobación de tipos de datos. No se tiene en cuenta para la corrección, por lo cual NO ES NECESARIA SU ENTREGA**)**

Además, en el directorio raíz , deberá aparecer el fichero **"nombres.txt"** : fichero de texto con los datos de los autores.

El formato de este fichero es:
1_DNI: *DNI1*
1_NOMBRE: *APELLIDO1.1 APELLIDO1.2, NOMBRE1*
2_DNI: *DNI2*
2_NOMBRE: *APELLIDO2.1 APELLIDO2.2, NOMBRE2*

(Los DNI sin guiones ni letras de NIF)

2.3. Entrega en formato DOXYGEN.

Comentarios para DOXYGEN

Para una correcta entrega en **formato Doxygen**, es necesario seguir las pautas que se indican el fichero "**Tutorial_DOXYGEN.pdf**", que se proporcionará como material de Campus Virtual.
Hay que incluir en los ficheros fuente todos los comentarios necesarios en **formato Doxygen**. Estos comentarios deben estar en forma corta y detallada, y deben definirse para :
- **Ficheros** : debe incluir **nombre** y **dni** de los autores
- **Clases** : escribir propósito de la clase: aprox. 3 líneas
- **Métodos (públicos o privados)** : 1 línea para funciones triviales. 2 líneas para operaciones específicas más complicadas (para éstas , explicar también : parámetros de entrada, parámetros de salida y funciones dependientes)

Generar documentación DOXYGEN

Para usar correctamente la **herramienta Doxygen**, es necesario colocar en el directorio raíz de la estructura de ficheros a entregar , el fichero de configuración **Doxygen** llamado "**DOXYFILE".**
Además, debe usarse un **MAKEFILE** adecuado para trabajar con **Doxygen** , de forma que cada vez que se ejecuta MAKE, se puede generar la documentación **Doxygen.**
Tanto el "**DOXYFILE"** como el **MAKEFILE** adecuado para **Doxygen** y para todos los TADs pedidos en la práctica, serán oportunamente publicados como materiales de Campus Virtual.
Cada vez que se ejecuta el **MAKEFILE** , **Doxygen** generará automáticamente un subdirectorio , **'doc'** , que contendrá la documentación de la práctica (extraída de los comentarios de los fuentes) , en formato HTML. Para ello, se ejecuta :

```
"make doc"
```

2.4 Entrega final

Sólo se deben entregar los ficheros detallados anteriormente (ninguno más).

Cuando llegue el momento de la entrega, toda la estructura de directorios ya explicada (¡ATENCIÓN! excepto el **MAKEFILE** , el **DOXYFILE** y la **carpeta 'doc'**, que hay que retirar antes), debe estar comprimida en un fichero de forma que éste NO supere los 300 K .

Ejemplo : **`tar -czvf PRACTICA.tgz *`**

2.5. Otros avisos referentes a la entrega .

No se devuelven las prácticas entregadas. Cada alumno es responsable de conservar sus prácticas.

La detección de prácticas similares (“copiados”) supone el automático suspenso de TODOS los autores de las prácticas similares. Cada alumno es responsable de proteger sus prácticas. Se recuerda que a los alumnos con prácticas copiadas no se les guardará ninguna nota (ni teoría ni prácticas), para convocatorias posteriores.

NOTA IMPORTANTE : Las prácticas no se pueden modificar una vez corregidas y evaluadas (no hay revisión del código). Por lo tanto, es esencial ajustarse a las condiciones de entrega establecidas en este enunciado.
En especial, llevar cuidado con los nombres de los ficheros.

ANEXO 3. Condiciones de corrección.

ANTES de la evaluación:

La práctica se programará en el Sistema Operativo Linux, y en el lenguaje C++. Deberá compilar con la versión **g++ 4.2.3** instalada en los laboratorios de la Escuela Politécnica Superior.

A lo largo del período cuatrimestral, el profesor irá realizando una REVISIÓN MANUAL , PERSONALIZADA e INDIVIDUAL del código de cada alumno. El resultado de esta revisión (como ya se expresa en las NORMAS DE LA ASIGNATURA), puede contabilizar HASTA 0'5 puntos (sobre 10) de la nota final de prácticas de PED; esto se evaluará en función de 2 factores :

a) Nivel de cumplimentación de los objetivos propuestos en fechas (ver apartado "**Cuándo se exige**" de cada clase propuesta)
b) Pruebas de calidad del código.

En el Campus Virtual se irán publicando ficheros de prueba (TADs) similares a los que se emplearán en la corrección. De todos modos, el alumno debe de crear su propio conjunto de ficheros de prueba para evaluar el funcionamiento de su práctica.

La evaluación:

La práctica se corregirá casi en su totalidad de un modo automático, por lo que los nombres de las clases, métodos, ficheros a entregar, ejecutables y formatos de salida descritos en el enunciado de la práctica SE HAN DE RESPETAR EN SU TOTALIDAD.

A la hora de la corrección del Examen de Prácticas (y por tanto, de la práctica del Cuadernillo) , se evaluará :
- El correcto funcionamiento de los TADs propuestos en en Cuadernillo
- El correcto funcionamiento de el/los nuevo/s método/s propuestos para programar durante el tiempo del Examen.

Uno de los objetivos de la práctica es que el alumno sea capaz de comprender un conjunto de instrucciones y sea capaz de llevarlas a cabo. Por tanto, es esencial ajustarse completamente a las especificaciones de la práctica.

Cuando se corrige la práctica , el corrector automático proporcionará ficheros de corrección llamados "**tad.cpp**". Este fichero utilizará la sintaxis definida para cada clase y los nombres de los ficheros asignados a cada una de ellas: únicamente contendrá una serie de instrucciones **#include** con los nombres de los ficheros "**.h**".

ANEXO 4. Utilidades

Almacenamiento de todos los ficheros en un único fichero

Usar el comando **tar** y **mcopy** para almacenar todos los ficheros en un único fichero y copiarlo en un disco:

- $ **tar** cvzf practica.tgz *
- $ **mcopy** practica.tgz a:/

Para recuperarlo del disco en la siguiente sesión:

- $ **mcopy** a:/practica.tgz
- $ **tar** xvzf practica.tgz

Cuando se copien ficheros binarios **(.tgz, .gif, .jpg**, etc.) no se debe emplear el parámetro -t en **mcopy**, ya que sirve para convertir ficheros de texto de Linux a DOS y viceversa.

Utilización del depurador gdb

El propósito de un depurador como **gdb** es permitir que el programador pueda "ver" qué está ocurriendo dentro de un programa mientras se está ejecutando.

Los comandos básicos de gdb son:

1. **r (run)**: inicia la ejecución de un programa. Permite pasarle parámetros al programa. Ejemplo: r fichero.txt.
2. **l (list)**: lista el contenido del fichero con los números de línea.
3. **b (breakpoint)**: fija un punto de parada. Ejemplo: b 10 (breakpoint en la línea 10), b main (breakpoint en la función main).
4. **c (continue)**: continúa la ejecución de un programa.
5. **n (next)**: ejecuta la siguiente orden; si es una función la salta (no muestra las líneas de la función) y continúa con la siguiente orden.
6. **s (step)**: ejecuta la siguiente orden; si es una función entra en ella y la podemos ejecutar línea a línea.
7. **p (print**): muestra el contenido de una variable. Ejemplo: p auxiliar, p this (muestra la dirección del objeto), p *this (muestra el objeto completo).
8. **h (help)**: ayuda.

Utilización de la herramienta VALGRIND

(Se proporcionará como Material complementario en Campus Virtual, un tutorial acerca de las características de la herramienta VALGRIND, que básicamente se usará para depuración de errores de código) .