Programación y Estructuras de Datos

CUADERNILLO 1

(Curso 2024-2025)

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 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.

Estructura de datos?
Modelo que resulta de definir los objetos y nociones de la realidad y de una manera organizada.

Acceder a los poros?
forma de obtener un objeto de la estructura de datos

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

Qué se pide:

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

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 un 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 volúmenes negativos en el **TPoro** están permitidos, imprimiéndolos con su signo negativo correspondiente.
- 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. Los acentos no se usarán en un color; ninguna prueba realizada utilizará caracteres con acentos.

se refiere a que se le pase un color en mayus? al constructor?

- El **Destructor** tiene que liberar toda la memoria que ocupe el objeto, asignando todos los valores numéricos a 0.
- En el "**operator==**", dos poros son iguales si poseen la misma posición, el mismo volumen y el mismo color.
- El operador SALIDA "**operator <<**" , muestra el contenido del **TPoro** . Características:
  - ✓ 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).

Redondeo para la variable "**volumen**": no se usará el redondeo, y sí las instrucciones, que actúan sobre el *output stream* (¡Atención! Si se fijan los decimales a mostrar, en ese TAD a partir de ese momento, cualquier número saldrá con ese número de decimales)

```
ostream & operator<<(ostream &os,const TPoro &poro){

  if(!poro.EsVacio()){
    os.setf(ios::fixed);
    os.precision( 2 );
    os<<"("<<poro.x<<", "<<poro.y<<") "<<poro.volumen<<" ";
    if(poro.color!=NULL)
      os<<poro.color;
    else
      os<<"-";
  }
  else
    os << "()";

  return os;
}
```

La precisión a 2 decimales solamente se tiene en cuenta en el operador SALIDA; para el resto de métodos la variable "**volumen** " es de tipo *double* sin más.

- ✓ 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.

Ejemplos de **operador SALIDA** :

```
(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.**

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" (es decir, con el valor directamente generado por el Constructor por defecto de **TPoro**).

- Se consideran elementos **TPoro** "vacíos", aquellos que contienen la posición igual a (0, 0), el volumen igual a 0 y el color con 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).

- El **Destructor** tiene que liberar toda la memoria que ocupe el vector, quedando la dimensión igual a 0.

- En el operador asignación "**operator=**", 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.

- En el operador igualdad "**operator==**", dos vectores son iguales si poseen la misma dimensión y los mismos
- elementos en las mismas posiciones.

- En la sobrecarga del operador corchete "**operator[]**", las posiciones van desde 1 (NO desde 0), hasta el tamaño o 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 método **"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 "**operator <<**" 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 **"[]"**.

  Ejemplos de **operador salida**:

  ```
  []

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

Parte 3: **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. Por tanto, también se pide construir una clase **TListaPosicion** que representa una posición en la lista de objetos de tipo **TPoro,** con su forma canónica (constructor, constructor de copia, destructor y sobrecarga del operador asignación) como mínimo.

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.

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

```
// Para implementar la POSICIÓN a NODO de la LISTA de TPoro
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();


// Extraer un rango de nodos de la lista
TListaPoro ExtraerRango (int n1, int n2)
```

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. NO es necesario comprobar que un **TListaPosicion** apunta realmente a un nodo de la lista (no se planteará el caso en los TAD de prueba); únicamente habrá que tener en cuenta que una posición puede ser vacía
- Sí que hay que comprobar (en concreto, para las operaciones "**Borrar"** y "**Obtener"**, propias de **TListaPoro**), que el objeto **TListaPosicion** no es vacío.
- En los métodos **"Anterior"** y **"Siguiente"**, si la posición actual es la primera o la última de la lista, se tiene que devolver una posición vacía.
- En el método " **EsVacia**" : se devuelve TRUE si el puntero interno ("**pos**") es NULL . En caso contrario devuelve FALSE.
- En el operador igualdad **"operator=="**, dos posiciones son iguales si apuntan a la misma posición de la lista.

Aclaraciones de la Clase "TListaPoro":

- Se permite AMISTAD entre las clases **TListaPosicion, TListaPoro y TListaNodo**.
- 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.
- En el operador asignación **"operator=",** 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 "**operator=="**, dos listas son iguales si poseen los mismos elementos en el mismo orden.
- El operador suma "**operator+",** une los elementos de dos listas en una nueva lista (ordenada y sin repetidos).

- El operador resta "**operator-"** 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 el método" **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.
  Esta regla se seguirá cumpliendo a medida que se puedan insertar **TPoros** con igual VOLUMEN.
  Devuelve TRUE si el elemento se puede insertar y FALSE en caso contrario (por ejemplo, porque ya exista en la lista).

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

- En el método "**Borrar(TListaPosicion &)"**, devuelve TRUE si el elemento se puede borrar (porque la posición apunta a un nodo de la lista) y FALSE en caso contrario (por ejemplo, porque la posición no es válida). El paso por referencia del parámetro es obligatorio, ya que una vez eliminado el elemento, la posición tiene que pasar a estar vacía (no asignada a ninguna lista).
  No es necesario comprobar que el objeto **TListaPosicion** apunte a un nodo de la lista.
  Hay que comprobar que el objeto **TListaPosicion** no es vacío

- En el método "**Obtener"**, si la posición está vacía se tiene que devolver un objeto **TPoro** "vacío".

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

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

- El operador SALIDA "**operator <<"** , 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 está vacía, se tiene que mostrar la cadena **"()"**

Ejemplo de **operador salida**:

Para este código…

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

…la salida sería :

```
((1, 1) 1.00 -)
```

- En el método **ExtraerRango :**

  Devuelve una lista con los elementos **TPoro** comprendidos entre las posiciones **n1 y n2** (ambas incluidas) de la lista que invoca a la función. Los nodos comprendidos entre **n1 y n2** (ambos incluidos) deben borrarse de la lista que invoca a la función.
  Cosas a tener en cuenta:

  - Se comienza a numerar las posiciones de la lista a partir de 1.
  - Si **n2** sobrepasa la longitud de la lista invocante "por exceso" : se seleccionan sólo los elementos contenidos entre **n1** y la longitud de la lista.
  - Si **n1** es menor o igual a 0: se seleccionan sólo los elementos contenidos entre la posición 1 de la lista y **n2** .
  - Si **n1 = n2** : devolverá una lista con 1 sólo elemento, extrayéndolo de la lista llamante.
  - Si **n1 > n2** : devolverá una lista VACÍA sin modificar a la llamante, pues los límites no engloban elemento alguno.

  Ejemplo *(el ejemplo está hecho con números naturales por simplificación):*

  Listas iniciales:

  ```
  L1=<1 4 6 7 8>
  L2=<>
  ```

  Llamada a la función:

  ```
  L2=L1.ExtraerRango(2,4)
  ```

  Listas finales después de ejecutar la función:

  ```
  L1=<1 8>
  L2=<4 6 7>
  ```

  Otra llamada a la función:

  ```
  L2=L1.ExtraerRango(4,2)
  ```

  Listas finales después de ejecutar la función:

  ```
  L1=<1 4 6 7 8>
  L2=<>
  ```

  Otra llamada a la función:

  ```
  L2=L1.ExtraerRango(0,18)
  ```

  Listas finales después de ejecutar la función:

  ```
  L1=<>
  L2=<1 4 6 7 8>
  ```

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

Cualquier modificación o comentario del enunciado se publicará oportunamente en UACLOUD.

En su momento, se publicarán como materiales del UACLOUD los distintos FICHEROS de MAIN (**main.cpp**) que servirán al alumno para realizar unas pruebas básicas con los MAIN propuestos.
No obstante, se recomienda al alumno que aporte sus propios ficheros **main.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 UACLOUD 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**: los métodos podrán opcionalmente devolver un mensaje de error (en **cerr**), solo en el caso de que 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.
En caso de controlar estas excepciones, 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).*

De cualquier modo, todos los métodos deben devolver siempre una variable del tipo que se espera.

ANEXO 2. Condiciones de ENTREGA.

2.1. Dónde, cómo, cuándo, valor.

La entrega de la práctica se realizará:

- Servidor: en el SERVIDOR DE PRÁCTICAS, cuya URL es : http://pracdlsi.dlsi.ua.es/

- Fecha: se habilitará la posibilidad de entrega en el Servidor de prácticas entre estas fechas:

  - **Lunes, 17/03/2025.**
  - **Viernes, 21/03/2025 (23:59)**

- A título INDIVIDUAL ; por tanto requerirá del alumno que conozca su USUARIO y CONTRASEÑA en el Servidor de Prácticas.

- Se podrá realizar cuantas entregas quiera el alumno: solo se corregirá la última práctica entregada. Tras cada entrega, el servidor enviará al alumno un **INFORME DE COMPILACIÓN**, para que el alumno compruebe que lo que ha entregado cumple las especificaciones pedidas y que se ha podido generar el ejecutable correctamente. Este informe también se podrá consultar desde la página web de entrega de prácticas del DLSI (http://pracdlsi.dlsi.ua.es e introducir el nombre de usuario y password). En caso de que la práctica esté correctamente entregada, compilada y ejecutada, en este informe debe salir lo siguiente:

  ```
  ================================================
  DIFERENCIA CON FICHERO DE SALIDA DE REFERENCIA
  ================================================

  ----------------------------------------
  ```

  Significa que la ejecución ha sido correcta y coincide con la salida esperada en el fichero de comprobación web publicado en UACLOUD.
  No es necesario entregar ni el makefile ni el main.cpp.
  Para la entrega y corrección se utilizará el makefile publicado en UACLOUD.

- Valor :
  - **El Cuadernillo 1 vale un 5% de la nota final de PED.**
  - El Cuadernillo 2 vale un 5% de la nota final de PED
  - La revisión presencial de prácticas vale un 15% de la nota final de PED.
  - El EXAMEN PRÁCTICO vale un 25% de la nota final de PED.

2.2. Ficheros a entregar y comprobaciones.

La práctica debe ir organizada en 3 subdirectorios:

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

DIRECTORIO 'lib': contiene los ficheros (NO deben entregarse los ficheros objeto "**.o**")**:**
- **"tporo.cpp"**
- **"tvectorporo. cpp "**
- **"tlistaporo. cpp " (**incluye clases**: TListaPoro , TListaNodo y TListaPosicion)**

DIRECTORIO 'src': contiene los ficheros:
- **"main.cpp" (**fichero aportado por el alumno para comprobación de tipos de datos. No se tiene en cuenta para la corrección**)**

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*

2.3 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**), debe estar comprimida en un fichero de forma que éste NO supere los 300 K (da igual el nombre del fichero .tgz que se entregue; al entrar en este .tgz deben aparecer SOLO los directorios y ficheros indicados anteriormente).

Ejemplo : `tar –czvf PRACTICA.tgz *`

El nombre del fichero **.tgz** entregado es indiferente. Debe contener SOLO los ficheros antes indicados.

2.4. 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. No está permitido usar código propio de la práctica extraído de cualquier medio (Internet, compañeros, etc.). La detección de una práctica copiada supone el suspenso automático en la prueba y el envío de un informe al depto. y a la EPS que tomarán medidas oportunas (Normativa EPS y Vicerrectorado de la UA).

ANEXO 3. Condiciones de corrección.

ANTES de la evaluación:

La práctica se programará en el Sistema Operativo Linux, y en el lenguaje C++. **Se corregirá con la versión de compilador de C++ instalada en los laboratorios de la Escuela Politécnica Superior.**

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 ficheros de prueba propuestos para el 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 "**main.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**".

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** y el **formato especificado para la salida.**

No está permitido usar código propio de la práctica, extraído de cualquier medio **(Internet, compañeros, etc.)**

ANEXO 4. Utilidades

Almacenamiento de todos los ficheros en un único fichero

Usar el comando **tar** para almacenar todos los ficheros en un único fichero:

- `$ tar cvzf practica.tgz *`

Para recuperarlo del disco en la siguiente sesión:

- `$ tar xvzf practica.tgz`

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 aconseja el uso de la herramienta VALGRIND para comprobar el manejo correcto de la memoria dinámica. Se puede encontrar una descripción más detallada de dicha herramienta en el libro **"C++ paso a paso"** de Sergio Luján.

Modo de uso. se recomienda usar este formato:

valgrind - -tool=memcheck <nombre_del_ejecutable>

Si se quiere usar en modo extenso, con información de **memoria no liberada** , se puede añadir otra directiva (no recomendado):

valgrind - -tool=memcheck - -leak-check=full <nombre_del_ejecutable>

Otras utilidades generales

El alumno tiene a su disposición en internet las siguientes URL con utilidades sobre C++, para consultar cualquier tipo de duda:

- **cplusplus** (www.cplusplus.com)
- **cppreference** (http://cppreference.com/