Programación y Estructuras de Datos
CUADERNILLO 1

(Curso 2016-2017)

Parte 1 : **clase base "TComplejo"**

Qué se pide :

Se pide construir una clase que representa un **NÚMERO COMPLEJO,** según su definición matemática.

Esta clase nos debe permitir realizar ARITMÉTICA COMPLEJA, partiendo de la definición básica de esta clase de números. Como debéis saber, los números complejos tienen esta forma :

(Parte REAL) + (PARTE IMAGINARIA) * i

Por lo tanto, necesitaremos 2 datos (usaremos para ellos el tipo DOUBLE) que nos permitan representar tanto la PARTE REAL como la PARTE IMAGINARIA.

Prototipo de la Clase:

PARTE PRIVADA

```
double  re ; // PARTE REAL

double  im ; // PARTE IMAGINARIA
```

FORMA CANÓNICA

```
//Constructor por defecto : PARTE REAL e IMAGINARIA  inicializadas a 0
TComplejo ()

//Constructor a partir de la PARTE REAL
TComplejo (double );

//Constructor a partir de la PARTE REAL e IMAGINARIA
TComplejo (double , double);

//Constructor copia
TComplejo (TComplejo&);

//Destructor
~TComplejo();

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

MÉTODOS

```
// SOBRECARGA DE OPERADORES ARITMÉTICOS;

TComplejo operator+ (TComplejo&);
TComplejo operator- (TComplejo&);
TComplejo operator* (TComplejo&);
TComplejo operator+ (double);
TComplejo operator- (double);
TComplejo operator* (double);

// OTROS OPERADORES
bool operator== (TComplejo&); // IGUALDAD de números complejos
bool operator!= (TComplejo&); // DESIGUALDAD de números complejos

double Re(); // Devuelve PARTE REAL
double Im(); // Devuelve PARTE IMAGINARIA

void Re(double); // Modifica PARTE REAL
void Im(double); // Modifica PARTE IMAGINARIA

double Arg(void); // Calcula el Argumento (en Radianes)
double Mod(void); // Calcula el Módulo
```

FUNCIONES AMIGAS

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

friend TComplejo operator+ (double , TComplejo&);
friend TComplejo operator- (double , TComplejo&);
friend TComplejo operator* (double , TComplejo&);
```

Aclaraciones :

- El **Constructor por defecto** inicializa la parte Real y la Imaginaria a 0**.**
- El **Constructor a partir de la parte Real** inicializa la parte Imaginaria a 0**.**
- El **Constructor por defecto** , el **Constructor a partir de la parte Real** y **Constructor a partir de la parte Real e Imaginaria** se podrían AGRUPAR en uno solo, síntesis de los tres, que sería **:**

  ```
  TComplejo (double = 0, double = 0);
  ```

- El **Destructor** deja la parte Real y la Imaginaria asignadas a 0**.**
- Para los operadores **ARITMÉTICOS,** es recomendable que los métodos que tengan un parámetro de entrada de tipo DOUBLE, operen de la siguiente manera :
  - Dentro del método, se convierte el valor DOUBLE en un objeto **TComplejo**, mediante el Constructor `TComplejo (double );`
  - Una vez hecho esto, el valor convertido ya puede operar con objeto de la clase, usando el operador correspondiente cuyo parámetro de entrada es un **TComplejo.**
- Para los operadores **ARITMÉTICOS**, téngase en cuenta que la definición del tipo DOUBLE en C++ admite la definición de un "cero negativo" (-0.0) y de un "cero positivo" (0.0). Dependiendo de cómo se realicen los cálculos entre variables de tipo DOUBLE, el resultado podría ser uno u otro. En cualquier caso, el valor aritmético de ambos es 0. Se debe tener en cuenta esta situación para convertir (-0.0) a (0.0). En las correcciones se tendrá en cuenta esta ambigüedad propia de este tipo de datos.
- Para los operadores **ARITMÉTICOS :**

  Ejemplos **:**

  ```
  TComplejo c1(3,4), c2(3,6), c3;

  c3 = c1 + c2;  /* ejemplo de uso del "TComplejo operator+ (TComplejo&)" → retorna c3 = (6,10)  */
  c3 = c1 + 7;  /* ejemplo de uso del "TComplejo operator+ (double);" → retorna c3 = (10,4) */
  c3 = 7 + c2;
  /* ejemplo de uso del " friend TComplejo operator+(…)" → retorna c3 = (10,6)  */
  ```

- El "**operator==**" establece que 2 números complejos son iguales si la parte Real es igual en ambos, y la parte Imaginaria es igual en ambos**.**
- El "**operator!=**" establece que 2 números complejos son iguales si la parte Real es diferente en ambos, o la parte Imaginaria es diferente en ambos**.**
- El método "**Arg**" debe devolver un valor comprendido en este rango : **[π, -π].**
- El método "**Mod**" se rige por la fórmula : **SQRT (POW(Re,2) + POW(Im,2))**
- Los métodos "**Arg**" y **"Mod"** deben de ser investigados por el alumno para resolver su implementación; baste decir que requerirán el uso de funciones matemáticas auxiliares (raíz cuadrada, arcotangente, etc.).

  Sobre la utilización del número PI (**π**) : se recomienda al alumno NO definir una constante cuyo valor sea 3.141597 ó similares, ya que, dependiendo de la precisión definida para el número, los resultados de ciertas operaciones podrían diferir, aunque sea ligeramente; por tanto, y para homogeneizar estas posibles discrepancias, se recomienda utilizar la función matemática "**atan2()".**

- Para que funcionen correctamente los métodos "**Arg"** y **"Mod"** de **TComplejo**, NO debe usarse instrucciones de este tipo al retornar el valor :

```
return sqrt(pow(re,2) + pow(im,2));
```

En cambio, se recomienda usar esta alternativa :

```
double a=sqrt(pow(re,2) + pow(im,2));
return a;
```

La razón reside en un problema de precisión de decimales.

Si los métodos "**Arg"** y **"Mod"** contienen este problema, pueden dar lugar a fallos en el correcto funcionamiento de los TADs de prueba de **TComplejo**, y además se arrastraría el problema para cuadernillos posteriores (árboles de búsqueda, etc.) .

- El **operador salida " operator <<"** , muestra por pantalla el contenido del número complejo en este formato :

Ejemplo de salida:

```
(3 1.55)
```

(Donde 3 corresponde a la PARTE REAL y 1.55 corresponde a la PARTE IMAGINARIA, separadas ambas por un " " , y todo el número englobado entre paréntesis, sin salto de línea detrás del cierre de paréntesis)

Parte 2 : **vector de TComplejo "TVectorCom"**

Qué se pide :

Se pide construir una clase que representa un **VECTOR de objetos de tipo "TComplejo"**.

Prototipo de la Clase:

PARTE PRIVADA

```
TComplejo *c;
int tamano;
```

FORMA CANÓNICA

```
// Constructor por defecto
TVectorCom ();
// Constructor a partir de un tamaño
TVectorCom (int);
// Constructor de copia
TVectorCom (TVectorCom &);
// Destructor
~TVectorCom ();
// Sobrecarga del operador asignación
TVectorCom & operator=( TVectorCom &);
```

MÉTODOS

```
// Sobrecarga del operador igualdad
bool operator==( TVectorCom &);
// Sobrecarga del operador desigualdad
bool operator!=( TVectorCom &);
// Sobrecarga del operador corchete (parte IZQUIERDA)
TComplejo & operator[](int);
// Sobrecarga del operador corchete (parte DERECHA)
TComplejo operator[](int) const;
// Tamaño del vector (posiciones TOTALES)
int Tamano();
// Cantidad de posiciones OCUPADAS (TComplejo NO VACIO) en el vector
int Ocupadas();
// Devuelve TRUE  si existe el TComplejo en el vector
bool ExisteCom(TComplejo &);
// Mostrar por pantalla los elementos TComplejo del vector  con PARTE REAL  IGUAL
O POSTERIOR al argumento
void MostrarComplejos(double);
// REDIMENSIONAR el  vector de TComplejo
bool  Redimensionar(int);
```

FUNCIONES AMIGAS

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

Aclaraciones :

- El vector NO tiene por qué estar ordenado.
- El vector puede contener elementos repetidos. Incluso de los considerados elementos **TComplejo** "vacíos"**.**
- Se consideran elementos **TComplejo** "vacíos", aquellos con valor **(0 0)**, por ejemplo los directamente generados por el **constructor por defecto de TComplejo.**
- El **Constructor por defecto** crea un vector de dimensión 0 (puntero interno a NULL, no se reserva memoria).
- En el **Constructor a partir de un tamaño**, si el tamaño es menor 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, dejándolo en un vector de dimensión 0.
- Si se asigna un vector a un vector no vacío, se destruye el vector inicial. Puede ocurrir que se modifique el tamaño 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 TComplejo en las mismas posiciones.
- En la sobrecarga del corchete, **"operator[]"**, las posiciones van desde 1 (no desde 0) hasta el tamaño del vector.

  Si se accede a una posición que no existe, se tiene que devolver un **TComplejo** "vacío" (el mismo que crea el **constructor por defecto** de **TComplejo**).

  Si se accede a una posición inexistente se debe devolver un **TComplejo** "vacío". Se permite declarar en la parte privada un elemento de clase de nombre "**error**" (para devolverlo por referencia)

- En "**MostrarComplejos (double)",** se muestra por pantalla cada uno de los elementos **TComplejo** de PARTE REAL IGUAL O POSTERIOR a la del argumento.
  El formato de salida será : la salida propia del elemento **TComplejo**, y cada **TComplejo** se separa del siguiente con una coma y espacio (", ") y toda la salida encerrada entre corchetes.
  A continuación del último elemento no se tiene que mostrar nada.
  Si no se ha de mostrar ningún **TComplejo**, la salida es corchetes vacíos "**[]**".

  Ejemplo:

```
TComplejo c1(1,0.5);
TComplejo c2(3,0.5);
TVectorComplejo v1(2);
v1[1] = c1;
v1[2] = c2;
v1.MostrarComplejos(1) ;
```

  (La salida sería):

```
[(1 0.5), (3 0.5)]
```

- En los **operadores "Ocupadas()" , "TVectorCom()" :**

  Se consideran "posiciones vacías" del vector, aquellas que contengan un **TComplejo** con valor **(0 0)**, por ejemplo los recién inicializados con el **Constructor por defecto** de **TComplejo**, o bien recién destruido con el **Destructor** de **TComplejo.**

- En el **operador "bool Redimensionar(int)" :**

  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 **TComplejo** inicializados con el **Constructor por defecto** de **TComplejo**.
  - ✓ Si el entero es mayor que 0 y menor que el tamaño actual del vector, se deben eliminar los **TComplejo** 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 sóla línea. Para cada elemento, primero se mostrará la POSICIÓN del elemento entre paréntesis y a continuación, separado por un espacio en blanco, el elemento **TComplejo** en sí. Cada elemento **TComplejo** se tiene que separar del siguiente por una coma y espacio (", ") .
  Toda la salida deber estar encerrada **entre corchetes "[]"**.
  NO se tiene que generar un salto de línea al final. Si la dimensión del vector es 0, se muestra la cadena **"[]".**

  Ejemplo :

```
TComplejo c1(1,0.5);
TComplejo c2(3,0.5);
TVectorCom v1(2);
v1[1] = c1;
v1[2] = c2;
cout << v1 ;
```

  (La salida sería) :

```
[(1) (1 0.5), (2) (3 0.5)]
```

Parte 3 : **lista de TComplejo " TListaCom "**

Qué se pide :

Se pide construir una clase que representa una **LISTA de objetos de tipo TComplejo,** que representa una lista DOBLEMENTE ENLAZADA y NO ORDENADA de números complejos accesible por una POSICIÓN**.**

No se permiten elementos repetidos.

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

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

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
TComplejo 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 “TListaPos”:

PARTE PRIVADA

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

FORMA CANÓNICA

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

// Constructor de copia
TListaPos (TListaPos &);

// Destructor
~TListaPos ();

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

MÉTODOS

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

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

// Devuelve la posición anterior
TListaPos Anterior();

// Devuelve la posición siguiente
TListaPos Siguiente();

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

Prototipo de la Clase “TListaCom”:

PARTE PRIVADA

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

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

FORMA CANÓNICA

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

// Constructor de copia
TListaCom (TListaCom &);

// Destructor
~TListaCom ();

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

MÉTODOS

```
// Sobrecarga del operador igualdad
bool operator==( TListaCom &);
// Sobrecarga del operador desigualdad
bool operator!=( TListaCom &);


// Sobrecarga del operador suma
TListaCom operator+(TListaCom &);
// Sobrecarga del operador resta
TListaCom operator-(TListaCom &);
// Devuelve true si la lista está vacía, false en caso contrario
bool EsVacia();
// Inserta el elemento en la cabeza de la lista
bool InsCabeza(TComplejo &);
// Inserta el elemento a la izquierda de la posición indicada
bool InsertarI(TComplejo &, TListaPos &);
// Inserta el elemento a la derecha de la posición indicada
bool InsertarD(TComplejo &, TListaPos &);
// Busca y borra la primera ocurrencia del elemento
bool Borrar(TComplejo &);
// Busca y borra todas las ocurrencias del elemento
bool BorrarTodos(TComplejo &);
// Borra el elemento que ocupa la posición indicada
bool Borrar(TListaPos &);
// Obtiene el elemento que ocupa la posición indicada
TComplejo Obtener(TListaPos &);
// Devuelve true si el elemento está en la lista, false en caso contrario
bool Buscar(TComplejo &);
// Devuelve la longitud de la lista
int Longitud();
// Devuelve la primera posición en la lista
TListaPos Primera();
// Devuelve la última posición en la lista
TListaPos Ultima();
```

FUNCIONES AMIGAS

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

Aclaraciones de la Clase “TListaPos” :

- 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).
- En cualquier caso, NO es necesario comprobar que un **TListaPos** 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 **Insertar**, **Borrar** y **Obtener**, propias de **TListaCom**), que el objeto **TListaPos** no es vacío.

- 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 “TListaCom” :

- Se permite AMISTAD entre las clases **TListaCom, TListaNodo y TListaPos.**

- La lista puede contener elementos repetidos

- La lista NO está ordenada.

- 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**, 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: primero los elementos de la primera lista (operando de la izquierda) y a continuación los elementos de la segunda lista (operando de la derecha).

- 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 **InsCabeza**, el nuevo elemento se inserta en la primera posición de la lista. Devuelve TRUE si el elemento se puede insertar y FALSE en caso contrario (por ejemplo, porque no se puede reservar memoria).

- En **Insertarl**, el nuevo elemento se inserta a la izquierda de la posición indicada. Devuelve TRUE si el elemento se puede insertar y FALSE en caso contrario (por ejemplo, porque no se puede reservar memoria). Hay que comprobar que el objeto **TListaPos** no es vacío.

- En **InsertarD,** el nuevo elemento se inserta a la derecha de la posición indicada. Devuelve TRUE si el elemento se puede insertar y FALSE en caso contrario (por ejemplo, porque no se puede reservar memoria). Hay que comprobar que el objeto **TListaPos** no es vacío.

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

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

- En **Obtener,** se tiene que devolver un número complejo creado con el constructor por defecto. Hay que comprobar que el objeto **TListaPos** no es vacío.

- En **Buscar**, se puede realizar una búsqueda lineal desde el primer elemento hasta el último.

- **Longitud** devuelve el número de nodos que hay en la lista.

- 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 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 lista está vacía, se tiene que mostrar la cadena "{}".

Ejemplo :

```
TComplejo c1(1,0.5);
TListaCom l1;
l1.InsCabeza(c1);
cout << v1 ;
```

(La salida sería) :

```
{(1 0.5)}
```

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, 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, 20/03/2017**
  - **Viernes, 24/03/2017 (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).
- Valor :
  - **El Cuadernillo 1 vale un 7% de la nota final de PED.**
  - El Cuadernillo 2 vale un 7% de la nota final de PED
  - El Cuadernillo 3 vale un 6% de la nota final de PED.
  - El EXAMEN PRÁCTICO vale un 30% 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) :
- **"tcomplejo.h"**
- **"tvectorcom.h"**
- **"tlistacom.h" (**incluye clases **: TListaCom , TListaNodo y TListaPos)**

DIRECTORIO 'lib' : contiene los ficheros (NO deben entregarse los ficheros objeto "**.o"**) **:**
- **"tcomplejo.cpp"**
- **"tvectorcom.cpp"**
- **"tlistacom.cpp" (**incluye clases **: TListaCom, TListaNodo y TListaPos)**

DIRECTORIO 'src' : contiene los ficheros :
- **"tad.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 (no se exige entregar el **MAKEFILE**), debe estar comprimida en un fichero de forma que éste NO supere los 300 K .

Ejemplo : `tar -czvf PRACTICA.tgz *`

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

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 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 TADs 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 "**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**".

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** 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 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:

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