UNIVERSIDAD DE MÁLAGA

E.T.S.I. TELECOMUNICACIÓN

Dpto. de Lenguajes y Ciencias de la Computación

**Programación 2**
(1° de Grados de SE, SI, ST, TT y Tel)
Junio 2016

| *Apellidos, Nombre* | *Ord* | *Grado* | *Grupo* | *Hora Entrega* |
|---|---|---|---|---|
| | | | | |

**Instrucciones para la realización del examen:**

- El código fuente de cada ejercicio debe guardarse en los ficheros especificados en el enunciado de cada ejercicio.
- Cada fichero debe comenzar con un comentario con el nombre, apellidos, ordenador, grado y grupo.
- Se deben respetar en todo momento los nombres especificados en el enunciado (nombres de ficheros, identificadores de tipos y de métodos), así como los prototipos para los métodos requeridos.
- **NO** se pueden añadir nuevos métodos públicos adicionales a las clases especificadas.
- **SI** se pueden añadir los métodos privados que el alumno considere necesarios a las clases especificadas.
- Los programas y clases resultantes deberán compilar correctamente y funcionar según lo especificado en el enunciado.
- Se valorará el código bien estructurado.
- Se debe obtener una **calificación superior (≥) a 3 puntos** en el **ejercicio 2** para que el resto de ejercicios puedan ser calificados.
- Al finalizar el examen:
  - Se deberán subir los ficheros conteniendo el código fuente de los ejercicios a la tarea especificada del *Campus Virtual* de la asignatura.
  - Además (salvo en los Mac), se deberán copiar los mismos ficheros al directorio de **Windows** [`Documentos\examen\p2jun16\`] (si no existe la ruta, deberá crearse).
  - Se deberá escribir la *hora de entrega* (además del resto de datos personales) en la cabecera del enunciado del examen y entregar al profesor.

---

### Ejercicio 1. La clase `Punto` (`punto.hpp` y `punto.cpp`) (1 pto.)

Un `Punto` describe una determinada posición en el *plano cartesiano*, especificada por el valor de las componentes $X$ e $Y$ (de tipo `double`) de las coordenadas cartesianas.

El *TAD* `Punto` (en el espacio de nombres `gps`) proporciona las siguientes operaciones públicas:

- Destructor: destruye el objeto `Punto` y todos sus recursos asociados.
- Constructor por defecto: construye un objeto `Punto` en el origen de coordenadas (0,0).
- Constructor específico: construye un objeto `Punto` según los valores de las coordenadas `cx` y `cy` recibidas como parámetros.
- Constructor de copia: construye un objeto `Punto` copiando sus valores de otro objeto `Punto` recibido como parámetro.
- Mostrar en pantalla las coordenadas $X$ e $Y$ del objeto actual, según el formato del siguiente ejemplo: `(3.5, 4.5)`.
- Desplazar el objeto `Punto` desde la posición actual una determinada distancia en ambos ejes `dx` y `dy` recibida como parámetro.
- Calcular la distancia absoluta, en el plano cartesiano, entre el objeto `Punto` actual y otro objeto `Punto` recibido como parámetro. $\delta = \sqrt{(X_1 - X_2)^2 + (Y_1 - Y_2)^2}$

- Devolver el valor de la componente $X$ del objeto `Punto` actual.
- Devolver el valor de la componente $Y$ del objeto `Punto` actual.

### Ejercicio 2. La clase `Ruta` (`ruta.hpp` y `ruta.cpp`) (6 ptos.)

Una `Ruta` describe un determinado camino como una secuencia de *Puntos* en el *plano cartesiano*.

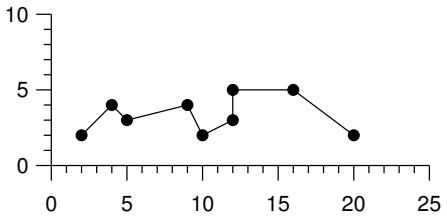
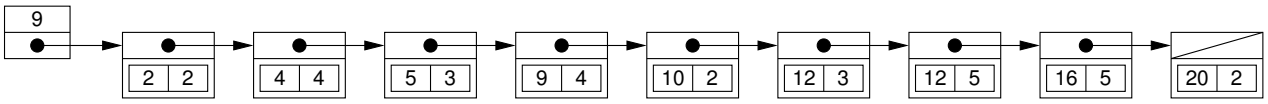



Figura 1: Lista de Puntos en el plano Cartesiano

El *TAD* `Ruta` (en el espacio de nombres `gps`) proporciona las siguientes operaciones públicas, considerando que los puntos del camino se almacenan como una *lista enlazada* secuencial con sus nodos en memoria dinámica, así como también se almacena la cantidad de puntos que componen el camino:




- Destructor: destruye el objeto `Ruta` actual y todos sus recursos asociados.
- Constructor por defecto: construye un objeto `Ruta` vacío, sin puntos almacenados.
- Constructor de copia: construye un objeto `Ruta` copiando sus valores de otro objeto `Ruta` recibido como parámetro.
- Mostrar en pantalla las coordenadas de la secuencia de puntos del camino del objeto actual, según el formato del siguiente ejemplo:

  ```
  [9]{ (2, 2), (4, 4), (5, 3), (9, 4), (10, 2), (12, 3), (12, 5), (16, 5), (20, 2) }
  ```

- Añadir un nuevo `Punto` recibido como parámetro, si no se encuentra previamente en la lista, al final de la lista enlazada de puntos que componen el camino. En otro caso, no se hará nada.
- Desplazar todos aquellos *Puntos* de la lista, cuyos componentes $X$ e $Y$ se encuentran dentro de una determinada área rectangular, desde sus posiciones actuales, una determinada distancia en ambos ejes `dx` y `dy` recibida como parámetro. El área rectangular se encuentra delimitada por las coordenadas de la esquina inferior izquierda y las coordenadas de la esquina superior derecha recibidas como parámetros (`minx`, `miny`, `maxx`, `maxy`).
- Calcular la distancia total recorrida, siguiendo secuencialmente todos los puntos de la `Ruta` actual. Si la ruta tiene menos de dos puntos, entonces devuelve cero (0).
- Comprobar si el camino del objeto actual contiene exactamente los mismos puntos, pero en orden **inverso**, que el camino contenido en otro objeto `Ruta` recibido como parámetro, en cuyo caso devuelve `true`, y en otro caso devuelve `false`.
- Devolver la cantidad de puntos que componen el camino del objeto `Ruta` actual.
- Devolver el objeto punto que se encuentra en una determinada posición, recibida como parámetro, de la secuencia en el camino de puntos del objeto `Ruta` actual, considerando que el primer punto de la secuencia se encuentra en la posición cero (0). Si el punto no existe, entonces se devolverá un punto en el origen de coordenadas (0,0).

- Eliminar, de la lista enlazada de puntos que componen el camino, en caso de que exista, un determinado `Punto` recibido como parámetro.
- Eliminar, según el orden de la secuencia, cada punto $P_i$ si existen los puntos $P_{i-1}$ y $P_{i+1}$ y la *distancia* entre $P_{i-1}$ y $P_i$ es menor que `dist` y la *distancia* entre $P_i$ y $P_{i+1}$ es menor que `dist` y la *distancia* entre $P_{i-1}$ y $P_{i+1}$ es menor que `dist`, considerando que `dist` es un valor `double` recibido como parámetro.

### Ejercicio 3. El *programa principal* (`main.cpp`) (2 ptos.)

El programa principal deberá realizar las siguientes acciones:

- Crear un objeto `r1` de clase `Ruta` y añadir secuencialmente los siguientes puntos, de tal forma que al mostrar el objeto `r1` muestre lo siguiente:

  `[9]{ (2, 2), (4, 4), (5, 3), (9, 4), (10, 2), (12, 3), (12, 5), (16, 5), (20, 2) }`
- Comprobar que si se intentan añadir puntos que ya se encuentran en el camino, entonces no se añaden realmente.
- Crear un objeto `r2` como copia (invocando al constructor de copia) del objeto `r1`, de tal forma que al mostrar el objeto `r2` muestre lo siguiente:

  `[9]{ (2, 2), (4, 4), (5, 3), (9, 4), (10, 2), (12, 3), (12, 5), (16, 5), (20, 2) }`
- Desplazar, una distancia de 3 y 5 en los ejes $X$ e $Y$ respectivamente, aquellos puntos de `r2` que están en el área delimitada por las coordenadas (7, 2.5) y (14, 4.5), de tal forma que al mostrar el objeto `r2` muestre lo siguiente:

  `[9]{ (2, 2), (4, 4), (5, 3), (12, 9), (10, 2), (15, 8), (12, 5), (16, 5), (20, 2) }`
- Calcular la distancia total recorrida en el objeto `r1`: 23.8379
- Calcular la distancia total recorrida en el objeto `r2`: 41.7952
- Consultar la cantidad de puntos del objeto `r1` (9) y los puntos de las posiciones 0, 4, 8, -1, 9:

  `(2, 2), (10, 2), (20, 2), (0, 0), (0, 0)`
- Crear un objeto `r3` de clase `Ruta` y añadir secuencialmente los siguientes puntos, de tal forma que al mostrar el objeto `r3` muestre lo siguiente:

  `[9]{ (20, 2), (16, 5), (12, 5), (12, 3), (10, 2), (9, 4), (5, 3), (4, 4), (2, 2) }`
- Comprobar que el camino contenido en la ruta `r1` **SI** es inverso al camino contenido en la ruta `r3`.
- Crear un objeto `r4` de clase `Ruta` y añadir secuencialmente los siguientes puntos, de tal forma que al mostrar el objeto `r4` muestre lo siguiente:

  `[9]{ (20, 2), (16, 5), (12, 5), (12, 3), (11, 3), (9, 4), (5, 3), (4, 4), (2, 2) }`
- Comprobar que el camino contenido en la ruta `r1` **NO** es el inverso al camino contenido en la ruta `r4`.
- Crear un objeto `r5` como copia (invocando al constructor de copia) del objeto `r1`, de tal forma que al mostrar el objeto `r5` muestre lo siguiente:

  `[9]{ (2, 2), (4, 4), (5, 3), (9, 4), (10, 2), (12, 3), (12, 5), (16, 5), (20, 2) }`
- Eliminar de `r5` los siguientes puntos (según el orden especificado):

  `(2, 2), (10, 2), (20, 2), (20, 5)`

  de tal forma que al mostrar el objeto `r5` muestre lo siguiente:

  `[6]{ (4, 4), (5, 3), (9, 4), (12, 3), (12, 5), (16, 5) }`

- Crear un objeto `r6` como copia (invocando al constructor de copia) del objeto `r1`, de tal forma que al mostrar el objeto `r6` muestre lo siguiente:

  ```
  [9]{ (2, 2), (4, 4), (5, 3), (9, 4), (10, 2), (12, 3), (12, 5), (16, 5), (20, 2) }
  ```

- Eliminar de `r6` aquellos puntos que cumplen la relación especificada anteriormente para una distancia de 7, de tal forma que al mostrar el objeto `r6` muestre lo siguiente:

  ```
  [5]{ (2, 2), (5, 3), (10, 2), (16, 5), (20, 2) }
  ```

**Ejercicio 4.** ***Programación Orientada a Objetos*** **(`main_videojuego.cpp`) (1 pto.)**

Desde la asignatura *Programación II* del campus virtual, pueden descargarse todos los ficheros de una jerarquía completa de clases polimórficas para el desarrollo de un videojuego. Los ficheros `objeto_espacial.hpp` y `objeto_espacial.cpp` desarrollan la clase `ObjetoEspacial`, que es la **clase base** de una jerarquía de herencia de objetos polimórficos . La clase `ObjetoEspacial` suministra, entre sus métodos, el método público `virtual escribir()`, que permite mostrar el objeto en pantalla.

De la clase `ObjetoEspacial` derivan las siguientes clases polimórficas: `Marciano` (`marciano.hpp`, `marciano.cpp`), `EspadaLaser` (`espada_laser.hpp`, `espada_laser.cpp`) y `NaveEspacial` (`nave_espacial.hpp`, `nave_espacial.cpp`), todas estas clases, derivadas de la clase base `ObjetoEspacial`, redefinen el método virtual `escribir()` de la clase base para mostrar en pantalla el objeto derivado correspondiente.

**Se pide**: desarrollar en el fichero `main_videojuego.cpp` un programa principal conteniendo lo siguiente:

- Un subprograma `pintar()`, que reciba como parámetro de entrada un puntero a la clase base de la jerarquía y, en caso de que este puntero no sea nulo, invoque al método `escribir()` sobre el objeto polimórfico de la jerarquía, recibido a través del puntero, para que se muestre por pantalla.
- Una función principal (`main`) que declare un *array* de 3 punteros a objetos de la clase base de la jerarquía de `ObjetoEspacial`, cree en memoria gestionada dinámicamente 3 objetos polimórficos distintos de la jerarquía (por ejemplo, un `Marciano`, una `EspadaLaser` y una `NaveEspacial`), y almacena sus referencias en posiciones distintas de *array*. Posteriormente, debe recorrerse el *array* para invocar al subprograma **`pintar()`** con cada uno de los objetos polimórficos incluidos en él. Finalmente, los objetos polimórficos creados deben destruirse antes de que el programa principal acabe.