Ejercicio 2:

- Diseñar una jerarquía con clase base 'Distancia' de herencia que permita sobrecargando el 'operador()' con parámetros a, b de tipo Punto, representar tres formas de medir la distancia entre dos puntos a y b:

1.- Distancia euclidea: float d = sqrt((a(x)-b(x))^2 + (a(y)-b(y))^2)
2.- Distancia manhattan: float d = abs(a(x)-b(x)) + abs(a(y)-b(y))
3.- Distancia chessboard: float d = min{abs(a(x)-b(x)), abs(a(y)-b(y)) }

- Implementar UNA función 'calcularDistancia' (no vale sobrecargarla para distintos tipos de distancias) que recibe tres parámetros: 'a' de tipo Punto, 'b' de tipo Punto y 'comp' un objeto de la jerarquía de herencia diseñada, y devuelve según el criterio de distancia implementado por 'comp', la distancia entre los puntos 'a' y 'b'.

NOTAS :

- Si no se sabe sobrecargar el operador(), utilizar la interfaz "dist" pero esto restará en la calificación.

- El fichero include para la jerarquía será 'distancias.hpp'

- Mirar el código de test.cpp para ver los nombres de las clases/funciones esperados.

ARCHIVOS ADJUNTOS

MAKEFILE

```
all: test

%.o : %.cc
	g++ -O0 -ggdb3 -Wall -c $< -o $@

test: clean test.o
	g++ -O0 -ggdb3 -Wall -o test *.o

.PHONY=clean
clean:
	rm -f test *.o *~
```

test.cc

```
#include <iostream>
#include <cassert>
```

```
#include <limits>
#include <cmath>
#include "punto.hpp"
#include "distancias.hpp"

inline bool
esIgual (float a, float b) throw ()
{
  return std::abs(a-b) <= std::numeric_limits<float>::epsilon();
}

int
main ()
{

  Punto a(0,0), b(1,1);

  std::cerr << "Comprobando distancia euclidea ...";
  assert (esIgual (calcularDistancia(a, b, DistanciaEuclidea()),
sqrt(2.0)));
  std::cerr << " ok." << std::endl;

  std::cerr << "Comprobando distancia manhattan ...";
  assert (esIgual (calcularDistancia(a, b,
DistanciaManhattan()), 2.0));
  std::cerr << " ok." << std::endl;

  std::cerr << "Comprobando distancia chessboard ...";
  assert (esIgual (calcularDistancia(a, b,
DistanciaChessboard()), 1.0));
  std::cerr << " ok." << std::endl;

  std::cerr << std::endl;
  std::cerr << "Enhorabuena, has pasado el test!!." <<
std::endl;

  return 0;
}
```

punto.h

```
#ifndef __PUNTO_HPP__
#define __PUNTO_HPP__

 class Punto
 {
  float _x, _y;
 public:
  Punto (): _x(0), _y(0) {}
  Punto (float x, float y) : _x(x), _y(y) {}
  float getX () const throw () { return _x ;}
  float getY () const throw () { return _y ;}
  void setX (float x) throw () { _x = x; }
```

```
  void setY (float y) throw () { _y = y; }
 };

#endif
```

distancias.h

```
//Pon aquÃ- tus declaraciones.

//Razonamiento:

//La clave es definir una clase base abstracta con el operador() como
//interface virtual pura, que despuÃ©s se harÃ¡ efectiva en cada clase derivada
//implementando una forma distancia concreta (euclÃ-dea, manhattan, ...).

//DespuÃ©s, la funciÃ³n calcularDistancia recibirÃ¡ una referencia del tipo
//base Distancia que estarÃ¡ conectada a un objeto cuyo tipo dinÃ¡mico serÃ¡
//una de las clases derivadas (segÃºn el tipo de distancia a calcular). Sobre
//esta referencia polimÃ³rfica se aplicarÃ¡ el operador '()' y,  debido
//a la ligradura dinÃ¡mica, se invocarÃ¡ la implementaciÃ³n de distancia
//correspondiente.

#include <cmath>
#include "punto.hpp"

class Distancia {
public:
 virtual float operator()(const Punto&a, const Punto&b) const throw () = 0;
};

class DistanciaEuclidea: public Distancia {
public:
 float operator()(const Punto&a , const Punto&b) const throw () {
  float xoff=b.getX()-a.getX();
  float yoff=b.getY()-a.getY();
  return sqrt(xoff*xoff + yoff*yoff);
 }
};

class DistanciaManhattan: public Distancia {
public:
 float operator()(const Punto&a , const Punto&b) const throw () {
  float xoff=b.getX()-a.getX();
  float yoff=b.getY()-a.getY();
  return std::abs(xoff) + std::abs(yoff);
```

```
  }
};

class DistanciaChessboard: public Distancia {
public:
  float operator()(const Punto&a , const Punto&b) const throw () {
    float xoff=b.getX()-a.getX();
    float yoff=b.getY()-a.getY();
    return std::max (std::abs(xoff), std::abs(yoff));
  }
};

inline
float calcularDistancia (const Punto&a, const Punto&b, const Distancia&comp)
  throw () {
  return comp(a, b);
}
```

Ejercicio3:

a) Unsando la clase Punto como modelo, define una plantilla denominada Punto_t donde el tipo base para las las coordenadas es genérico para poder así tener puntos en el dominio de los enteros y reales.

b) Define una función plantilla 'maximo' con dos argumentos a, b del mismo tipo que devuelve el valor máximo (usando el operador <) de los argumentos.

Especializa esta plantilla de función para el tipo Punto_t para floats, devolviendo el punto que tenga el módulo (sqrt(x*2+y*2)) más grande.

Especializa esta plantilla de función para el tipo Punto_t para enteros, devolviendo el punto que tenga la suma en valor absoluto de sus componentes mayor (|x| + |y|) más grande.

NOTAS:

- Mirar el código de test.cc para ver los identificadores esperados para includes y clases.

- La función valor absoluto para enteros (abs) está definida en stdlib.h y no en math.h

ARCHIVOS ADJUNTOS

MAKEFILE

```
all: test

%.o : %.cc
	g++ -O0 -ggdb3 -Wall -c $< -o $@

test: clean test.o
	g++ -O0 -ggdb3 -Wall -o test *.o

.PHONY=clean
clean:
	rm -f test *.o *~
```

test.cc

```
#include <iostream>
#include <cassert>

#include "punto_t.hpp"


int
main (void)
{
  Punto_t<float> a (1, 1), b (1.5, 1.5);
  Punto_t<int> c (1, 1), d (2, 2);

  std::cerr << "Comprobando especializacion de maximo para
floats ... ";
  Punto_t<float> maxF = maximo (a, b);
  assert (maxF.getX() == b.getX() and maxF.getY() == b.getY());
  std::cerr << "ok." << std::endl;

  std::cerr << "Comprobando especializacion de maximo para
enteros ... ";
  Punto_t<int> maxI = maximo (c, d);
  assert (maxI.getX() == d.getX() and maxI.getY() == d.getY());
  std::cerr << "ok." << std::endl;

  return 0;
}
```

punto.h

```
#ifndef __PUNTO_HPP__
#define __PUNTO_HPP__

 class Punto
 {
```

```
    float _x, _y;
  public:
    Punto (): _x(0), _y(0) {}
    Punto (float x, float y) : _x(x), _y(y) {}
    float getX () const throw () { return _x ;}
    float getY () const throw () { return _y ;}
    void setX (float x) throw () { _x = x; }
    void setY (float y) throw () { _y = y; }
  };

#endif
```

Punto_T.h

```
/* -*- conding: utf-8 -*- */
//AquÃ- debes poner tu codigo de plantillas
#ifndef __PUNTO_T_HPP__
#define __PUNTO_T_HPP__

#include <cmath>
#include <cstdlib>

//Para el template de la clase punto basta copiarla y cambiar toda
//referencia a float por el tipo generÃ©rico, por ejemplo G
//(buscar/reemplazar del navegador)

template<class G>
  class Punto_t
  {
    G _x, _y;
  public:
    Punto_t (): _x(0), _y(0) {}
    Punto_t (G x, G y) : _x(x), _y(y) {}
    G getX () const throw () { return _x ;}
    G getY () const throw () { return _y ;}
    void setX (G x) throw () { _x = x; }
    void setY (G y) throw () { _y = y; }
  };

//Definir una funciÃ³n plantilla para obtener el mÃ¡ximo de dos valores
//usando el operador < es fÃ¡cil...
template <class G>
G maximo (const G& a, const G& b) throw ()
{
  return (a<b) ? b : a;
```

```
}

//Para especializar una plantilla C++ tiene una sintÃ¡xis concreta
//No vale decir por ejemplo:
//
//inline
//Punto_t<float> maximo (const Punto_t<float>& a, const Punto_t<float>& b)
//{...}
//
//Ya que esto realmente indica una SOBRECARGA y no una especializaciÃ³n.
//La forma correcta de especializar es la siguiente:
//
//        -------------------Esta es la clave de la especializaciÃ³n.
//        |           |
//        |           |
//        V           V
//template <> TS maximo<TS> (const TS& a, const TS& b) throw();
//
//donde TS es el tipo para el que se especializa la plantilla.
//Otra cosa a quidar es cuando ponemos el tipo Punto_t<float> por ejemplo
// ya que si ponemso maximo<Punto_t<float>> el Ãºltimo >> confunde a C++ con
//operador '>>' asÃ- que hay que separar con un espacio blanco:
// maximo<Punto_t<float> > ... OK!
//

template <>
Punto_t<float> maximo < Punto_t<float> > (const Punto_t<float> &a,
                                          const Punto_t<float> &b) throw ()
{
  float aMod = sqrt(a.getX()*a.getX()+a.getY()*a.getY());
  float bMod = sqrt(b.getX()*b.getX()+b.getY()*b.getY());
  return (aMod>bMod) ? a : b;
}


template <>
Punto_t<int> maximo < Punto_t<int> > (const Punto_t<int> &a,
                                      const Punto_t<int> &b) throw ()
{
  float aMod = abs(a.getX())+abs(a.getY());
  float bMod = abs(b.getX())+abs(b.getY());
  return (aMod>bMod) ? a : b;
}
```

```
#endif //__PUNTO_T_HPP__
```