PRÁCTICA 6
EJERCICIO 9

ICOM

PRÁCTICA 6 – EJERCICIO 9

Escriba un programa que utilizando la siguiente estructura para representar puntos en 2D:

```
struct Punto2D {
    double x;
    double y;
};

struct Triangulo {
    Punto2D vertices[3];

    enum ClaseLado { EQUILATERO, ISOCELES, ESCALENO };
    enum ClaseAngulo { ACUTANGULO, RECTANGULO, OBTUSANGULO };

    ClaseLado clasificaPorLado();

    ClaseAngulo clasificaPorAngulo();
};
```

a) Implemente los métodos `clasificaPorLado()` y `clasificaPorAngulo()`.

b) Realice un programa que pida al usuario los puntos correspondientes a los 3 vértices de un triángulo y pruebe los métodos anteriores.

PUNTO B)

```
#include "icom_helpers.h"

int main()
{
    Triangulo tr;

    cout << "Ingrese 3 puntos 2D que definen un triangulo (3x [X Y]) ";
    tr.ingresa();

    cout << "El triangulo es: ";
    tr.print();
    cout << endl;

    cout << "Clase por lado: " << tr.clasificaPorLado() << endl;

    cout << "Clase por angulo: " << tr.clasificaPorAngulo() << endl;

    return 0;
}
```

ingresa() Y print()

```
struct Punto2D {
    double x;
    double y;

    void ingresa() {
        cin >> x >> y;
    }

    void print() {
        cout << '(' << x << ','
             << y << ')';
    }
};
```

```
struct Triangulo {
      Punto2D vertices[3];

      void ingresa() {
          vertices[0].ingresa();
          vertices[1].ingresa();
          vertices[2].ingresa();
      }

      void print() {
          cout << "{ ";
          vertices[0].print();
          cout << ' ';
          vertices[1].print();
          cout << ' ';
          vertices[2].print();
          cout << " }";
       }
  };
```

PUNTO A) FUNCIONES AUXILIARES DE `Punto2D`

```
struct Punto2D {
    double x;
    double y;

    Punto2D operator+(Punto2D a) {
        return Punto2D{ x + a.x, y + a.y };
    }

    Punto2D operator-(Punto2D a) {
        return Punto2D{ x - a.x, y - a.y };
    }

    double sqabs() {
        return x * x + y * y;
    }

    double abs() {
        return sqrt(sqabs());
    }

    double dot(Punto2D a) {
        return x * a.x + y * a.y;
    }
};
```

PUNTO A) `clasificaPorLado()`

```
struct Triangulo {
    Punto2D vertices[3];
    enum ClaseLado { EQUILATERO, ISOCELES, ESCALENO };

    ClaseLado clasificaPorLado() {
        Punto2D lados[] = { vertices[1] - vertices[0], vertices[2] - vertices[1], vertices[0] - vertices[2] };
        double largos_sq[] = { lados[0].sqabs(), lados[1].sqabs(), lados[2].sqabs() };

        if(    largos_sq[0] == largos_sq[1]
            && largos_sq[0] == largos_sq[2] ) {
            return EQUILATERO;
        }

        if(    largos_sq[0] == largos_sq[1]
            || largos_sq[1] == largos_sq[2]
            || largos_sq[2] == largos_sq[0] ) {
            return ISOCELES;
        }
        return ESCALENO;
    }
};
```

PUNTO A) `clasificaPorLado()`

```
bool es_aprox_igual(double a, double b, double tol = 1e-6) {
   return fabs(a-b) < tol ? true : false;
}

struct Triangulo {
    Punto2D vertices[3];
    enum ClaseLado { EQUILATERO, ISOCELES, ESCALENO };

    ClaseLado clasificaPorLado() {
        Punto2D lados[] = { vertices[1] - vertices[0], vertices[2] - vertices[1], vertices[0] - vertices[2] };
        double largos_sq[] = { lados[0].sqabs(), lados[1].sqabs(), lados[2].sqabs() };

        if(    es_aprox_igual(largos_sq[0], largos_sq[1])
            && es_aprox_igual(largos_sq[0], largos_sq[2]) ) {
            return EQUILATERO;
        }

        if(    es_aprox_igual(largos_sq[0], largos_sq[1])
            || es_aprox_igual(largos_sq[1], largos_sq[2])
            || es_aprox_igual(largos_sq[2], largos_sq[0]) ) {
            return ISOCELES;
        }
        return ESCALENO;
    }
};
```

es_aprox_igual() MEJORADA

```
bool es_aprox_igual(double a, double b, double tol = 1e-6)
{
   double dif = fabs(a-b);

   if( a != 0 )
       dif /= a;
   else if( b != 0 )
       dif /= b;

   return dif < fabs(tol) ? true : false;
}
```

PUNTO A) `clasificaPorLado()`

```
bool es_aprox_igual(double a, double b, double tol = 1e-6) {
   return fabs(a-b) < tol ? true : false;
}

struct Triangulo {
    Punto2D vertices[3];
    enum ClaseAngulo { ACUTANGULO, RECTANGULO, OBTUSANGULO };

    ClaseAngulo clasificaPorAngulo() {
        Punto2D lados[] = { vertices[1] - vertices[0], vertices[2] - vertices[1], vertices[0] - vertices[2] };
        double dot_prods[] = { lados[0].dot(lados[1]), lados[1].dot(lados[2]), lados[2].dot(lados[0]) };

        if(    es_aprox_igual(dot_prods[0], 0)
            || es_aprox_igual(dot_prods[1], 0)
            || es_aprox_igual(dot_prods[2], 0) ) {
            return RECTANGULO;
        }

        if(    dot_prods[0] < 0
            || dot_prods[1] < 0
            || dot_prods[2] < 0 ) {
            return OBTUSANGULO;
        }

        return ACUTANGULO;
};
```

PUNTO B)

```
#include "icom_helpers.h"

int main()
{
    Triangulo tr;

    cout << "Ingrese 3 puntos 2D que definen un triangulo (3x [X Y]) ";
    tr.ingresa();

    cout << "El triangulo es: ";
    tr.print();
    cout << endl;

    cout << "Clase por lado: " << tr.clasificaPorLado() << endl;

    cout << "Clase por angulo: " << tr.clasificaPorAngulo() << endl;

    return 0;
}
```

```
Ingrese 3 puntos 2D que definen un triangulo (3x [X Y])
0 0 1 0 0 1
El triangulo es: { (0,0) (1,0) (0,1) }
Clase por lado: 1
Clase por angulo: 1
```

IMPRIMIR NOMBRE DE LA ENUMERACIÓN

```
string to_string(Triangulo::ClaseLado cl) {
    switch( cl ) {
        case Triangulo::EQUILATERO:
            return "EQUILATERO";

        case Triangulo::ISOCELES:
            return "ISOCELES";

        case Triangulo::ESCALENO:
            return "ESCALENO";
    }
    // return "ClaseLado_INVALIDO";
    throw( runtime_error("is_string: Triangulo::ClaseLado INVALIDO");
}

int main() {
    // ...

    Triangulo::ClaseLado cl = tr.clasificaPorLado();
    cout << "Clase por lado: " << to_string(cl) << " (" << cl << ")\n";

    // ...

}
```

IMPRIMIR NOMBRE DE LA ENUMERACIÓN

```
struct Triangulo {
    // ...

    static string to_string(ClaseAngulo ca) {
        switch( ca ) {
            case ACUTANGULO:
                return "ACUTANGULO";

            case RECTANGULO:
                return "RECTANGULO";

            case OBTUSANGULO:
                return "OBTUSANGULO";
        }

        // return "ClaseAngulo_INVALIDO";
        throw( runtime_error("Triangulo::is_string: ClaseAngulo INVALIDO");
}

int main() {
    // ...

    Triangulo::ClaseAngulo ca = tr.clasificaPorAngulo();
    cout << "Clase por angulo: " << Triangulo::to_string(ca) << " (" << ca << ")\n";

    // ...

}
```

PUNTO B)

```
#include "icom_helpers.h"

int main()
{
    Triangulo tr;

    cout << "Ingrese 3 puntos 2D que definen un triangulo (3x [X Y]) ";
    tr.ingresa();

    cout << "El triangulo es: ";
    tr.print();
    cout << endl;

    Triangulo::ClaseLado cl = tr.clasificaPorLado();
    cout << "Clase por lado: " << to_string(cl) << " (" << cl << ")\n";

    Triangulo::ClaseAngulo ca = tr.clasificaPorAngulo();
    cout << "Clase por angulo: " << Triangulo::to_string(ca) << " (" << ca << ")\n";

    return 0;
}
```

```
Ingrese 3 puntos 2D que definen un triangulo (3x [X Y])
0 0 1 0 0 1
El triangulo es: { (0,0) (1,0) (0,1) }
Clase por lado: ISOCELES (1)
Clase por angulo: RECTANGULO (1)
```