UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

FACULTAD DE INGENIERIA DE PRODUCCION Y SERVICIOS

ESCUELA PROFESIONAL DE CIENCIAS DE LA COMPUTACIÓN

LABORATORIO 20 – Smart Pointers.

DOCENTE:

Enzo Edir Velásquez Lobatón

ALUMNO:

Owen Haziel Roque Sosa.

FECHA:

14/08/2022

Arequipa – Perú

1. Implemente el siguiente código que usa punteros sin procesar y explique lo que hace:

```
{
        double* d = new double(1.0);
        Point* pt = new Point(1.0, 2.0);

        *d = 2.0;
        (*pt).X(3.0);
        (*pt).Y(3.0);

        pt->X(3.0);
        pt->Y(3.0);

        delete d;
        delete pt;
}
```

```
// Implementacion de la clase Point
class Point {
private:
    double x, y;
public:
    Point(double x_axis, double y_axis) : x(x_axis), y(y_axis) {}
    void print() const {
        cout << "(" << x << "," << y << ")" << endl;
    }
    ~Point() {
        cout << "Punto destruido" << endl;
    }
    void X(double x) {
        this->x = x;
    }
    void Y(double y) {
        this->y = y;
    };
};
```

- Se crean punteros a double y a objeto Point, con sus respectivos valores.
- Se hace una dereferencia y se llaman a las funciones miembro (métodos) de la clase Point (o struct, depende).
  - Se modifican las coordenadas X, Y a través del operador de indirección (*)
- Se modifican las coordenadas X, Y a través del mismo puntero (pt)
- Se limpia la memoria a través de *delete.*

2. Transfiera el código anterior reemplazando los punteros sin formato por std::unique_ptr.

```
void ejercicio() {
    std::unique_ptr<double> d = std::make_unique<double>(1.0);
    std::unique_ptr<Point> pt = std::make_unique<Point>(1.0, 2.0);
    *d = 2.0;
    (*pt).X(3.0);
    (*pt).Y(3.0);
    pt->X(3.0);
    pt->Y(3.0);
//  delete d;
//  delete pt;
}

int main(int argc, char *argv[]) {
    ejercicio();
    return 0;
}
```

```
Punto destruido


<< El programa ha finalizado: codigo de salida: 0 >>
<< Presione enter para cerrar esta ventana >>
```

3. Implementar el código para las clases C1 y C2, cada una de las cuales contiene el objeto compartido d anterior, por ejemplo:

```
class C1
        {
        private:
                std::shared_ptr<double> d;
        public:
                C1(std::shared_ptr<double> value) : d(value) {}
                virtual ~C1() { cout << "\nC1 destructor\n"; }
                void print() const { cout << "Valor " << *d; }
        };
```

```
#include <iostream>
#include <memory>
using namespace std;

class C1 {
private:
    std::shared_ptr<double> d;
public:
    C1(std::shared_ptr<double> value) : d(value) {}
    virtual ~C1() { std::cout << "\nC1 destructor\n";}
    void print() const {
        cout << "Valor: " << *d;
    }
};

class C2 {
private:
    std::shared_ptr<double> d;
public:
    C2(std::shared_ptr<double> value) : d(value) {}
    virtual ~C2() { std::cout << "\nC2 destructor\n";}
    void print() const {
        cout << "Valor: " << *d;
    }
};
```

4. Transfiera el código anterior reemplazando los punteros sin formato por std::shared_ptr<Point> p;

```cpp
#include <iostream>
#include <memory>
using namespace std;

class Point {
private:
    double x, y;
public:
    Point(double x_axis, double y_axis) : x(x_axis), y(y_axis) {}
    void print() const {
        cout << "(" << x << "," << y << ")" << endl;
    }
    ~Point() {
        cout << "\nPunto " << "(" << x << "," << y << ")" << " destruido\n" << endl;
    }
    void X(double x) {
        this->x = x;
    }
    void Y(double y) {
        this->y = y;
    };
};

class C1 {
private:
    std::shared_ptr<Point> d;
public:
    C1(std::shared_ptr<Point> value) : d(value) {
        cout << "\nC1 constructor\n";
    }
    virtual ~C1() { std::cout << "\nC1 destructor\n";}
    void print() const {
        d->print();
    }
};
```

```
class C2 {
private:
    std::shared_ptr<Point> d;
public:
    C2(std::shared_ptr<Point> value) : d(value) {
        cout << "\nC2 constructor\n";
    }
    virtual ~C2() { std::cout << "\nC2 destructor\n";}
    void print() const {
        d->print();
    }
};

int main(int argc, char* argv[]) {
    std::shared_ptr<Point> p(new Point(5.0,6.7));
    C1 c1(p);
    cout << "C1 -> ";   c1.print();
    C2 c2(p);
    cout << "C2 -> ";   c2.print();
    return 0;
}
```

```
C1 constructor
C1 -> (5,6.7)

C2 constructor
C2 -> (5,6.7)

C2 destructor

C1 destructor

Punto (5,6.7) destruido



<< El programa ha finalizado: codigo de salida: 0 >>
<< Presione enter para cerrar esta ventana >>
```

5. Al anterior código implemente un puntero débil a un puntero el cual no puede estar vacío.

- Mismas Clases Point, C1, C2 del ejercicio anterior

```
int main(int argc, char* argv[]) {
    std::shared_ptr<Point> p(new Point(5.0,6.7));
    // En el anterior ejercicio se usa shared_ptr a
    // objeto Point, así que se usara weak_ptr al
    // shared_ptr anterior
    std::weak_ptr<Point> wp;
    wp = p;
    C1 c1(p);
    cout << "C1 -> ";   c1.print();
    C2 c2(p);
    cout << "C2 -> ";   c2.print();
    cout << "\n Cant. Objetos shared_ptr que gestiona wp = "
         << wp.use_count() << endl;
    return 0;
}
```

```
C1 constructor
C1 -> (5,6.7)

C2 constructor
C2 -> (5,6.7)

 Cant. Objetos shared_ptr que gestiona wp = 3

C2 destructor

C1 destructor

Punto (5,6.7) destruido



<< El programa ha finalizado: codigo de salida: 0 >>
<< Presione enter para cerrar esta ventana >>
```

Con la función use_count() de std::weak_ptr se verifica la cantidad de objetos std::shared_ptr gestionados por *wp*, que son 3:

- El std::shared_ptr<Point> p Inicial (definido en main),
- std::shared_ptr<Point> d de la clase C1,
- std::shared_ptr<Point> d de la clase C2