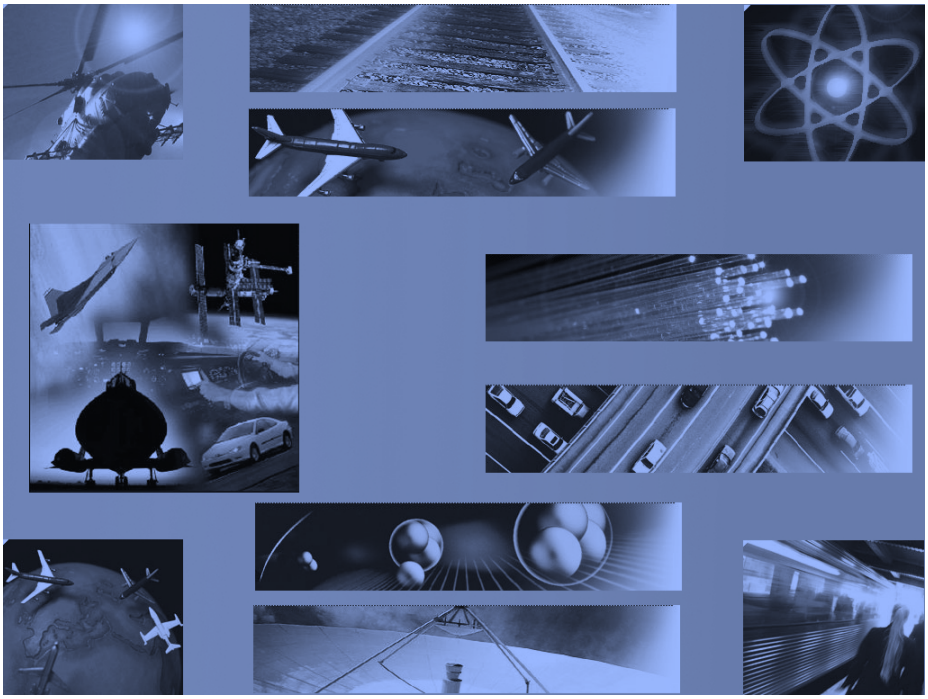
EXAMEN JUNIO 2022
PROGRAMACIÓN DE SISTEMAS DE TIEMPO-REAL



GRADO DE INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA

industriales
etsii UPCT

Universidad Politécnica de Cartagena

EJERCICIO 1

Se pide que diseñe parte de la infraestructura de una aplicación de dibujo vectorial partiendo del siguiente código:

```
struct I_Forma {
    virtual double calcular_perimetro() const = 0;
    virtual double calcular_area() const = 0;
    friend ostream& operator<< (ostream &os, const I_Forma &f);
    I_Forma (int id) : id_{id} {}
    int get_id() const { return id_; }
protected :
    virtual void print(ostream &os) const = 0;
private:
    int id_;
};
ostream& operator<< (ostream &os, const I_Forma &f) {
    f.print(os);
    return os;
}
```

Se pide que implemente las siguientes clases, que comparten una jerarquía de herencia:

Triangulo

- Hereda de I_Forma.
- Tiene como **atributos** las longitudes de los tres lados que lo forman.
- Tiene un **constructor** que recibe como parámetros el identificador y las longitudes de los lados que lo forman.
- Implementa los tres **métodos virtuales** de I_Forma. Para el cálculo del área, se utiliza la fórmula de Herón: $\sqrt{s \cdot (s - l_1) \cdot (s - l_2) \cdot (s - l_3)}$, donde 's' es el semiperímetro (la mitad del perímetro del triángulo). Una posible implementación del método print podría generar: "*Triangulo #1, l1=3, l2=4, l3=6, area: 5.33268, perimetro: 13*".

Rectángulo

- Hereda de I_Forma.
- Tiene como **atributos** las longitudes de los dos lados que lo forman.
- Tiene un **constructor** que recibe como parámetros el identificador y las longitudes de los lados que lo forman.
- Implementa un método **get_lado1**.
- Implementa los tres **métodos virtuales** de I_Forma. Una posible implementación del método print podría generar: "*Rectangulo #2, l1=3, l2=4, area: 12, perimetro: 14*".

Cuadrado

- Hereda de Rectangulo.
- Tiene como **atributo** la longitud del lado que lo forma.
- Tiene un **constructor** que recibe como parámetros el identificador y la longitud del lado.

- Implementa solo el **método virtual** print, que podría generar: "*Cuadrado #3 de lado=3, area: 9, perimetro: 12*".

Escriba una función **main** de prueba corta para comprobar el funcionamiento de las clases desarrolladas y que el polimorfismo funciona adecuadamente. Para ello, cree un objeto de cada clase, lo asigna a un puntero de tipo I_Forma y luego lo imprime por consola a través del puntero. Solo puede utilizar la funcionalidad que proporciona la librería estándar de C++.

EJERCICIO 2

Se pide programar la infraestructura básica para gestionar grafos no dirigidos. Un grafo está definido por tener un conjunto de nodos y un conjunto de arcos, que conectan entre sí dos nodos. Estos arcos pueden definir una dirección para la conexión o ser bidireccionales (no dirigidos). Se pide que defina una estructura Nodo, que almacena el identificador (un número entero) del nodo, y una clase Grafo, con el siguiente contenido:

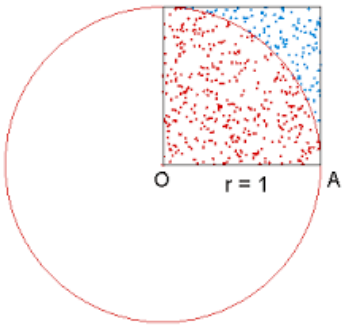
- Como **`atributos`**, tiene un vector de Nodos y un mapa que asocia cada Nodo con el conjunto de nodos con que está conectado. Para evitar replicar nodos, el mapa es de punteros a Nodo. Tiene también el número de arcos del grafo.
- Método **`add_nodo()`**, que recibe como parámetro un nodo y la añade al vector de nodos si no existe ya un nodo con el mismo identificador. Retorna un valor booleano para indicar si ha añadido el nodo o no.
- Método **`add_arco()`**, que recibe como parámetros los códigos de identificación de dos nodos y retorna un valor lógico que indica si ha podido crear el arco entre ambos nodos. El arco se puede crear si los códigos de los nodos son distintos, si existen los nodos y si no existía ya un arco entre ambos nodos. Si se cumplen estas restricciones, se añade el arco en ambos nodos del mapa (por ejemplo, `add_arco(1,2)` añade el nodo 2 al conjunto de nodos con que está conectado el nodo 1, y añade el nodo 1 al conjunto de nodos con que está conectado el nodo 2).
- Método **`max_num_arcos_nodo()`**, que retorna cuál es el número máximo de arcos que tiene un nodo del grafo.
- El **`operator<<`** para imprimir el contenido del grafo por consola. Por ejemplo:

```
Grafo con 6 nodos y 7 arcos
Nodo 1
      2    3    4    6
Nodo 2
      1    4    6
... el resto de nodos
```

Escriba una función **main** de prueba corta para comprobar el funcionamiento de las clases desarrolladas. Solo puede utilizar la funcionalidad que proporciona la librería estándar de C++.

EJERCICIO 3: CONCURRENCIA

Se quiere realizar un programa que realice el cómputo del valor de PI mediante el método de Monte Carlo. Este método utiliza valores aleatorios para resolver algunos problemas, sobre todo de simulación. En este caso, el cálculo de PI se realiza de la siguiente manera: se construye un cuadrado de lado 1 y se inscribe un cuarto de una circunferencia de lado 1, centrada en el vértice inferior izquierdo. Se genera un conjunto de pares x-y de forma aleatoria, dentro del cuadrado, que representan la coordenada de un punto. Se cuentan cuantos puntos se generan dentro de la circunferencia. Se multiplica este número por 4 y se divide por el total de pares generados y se obtiene la estimación de PI.

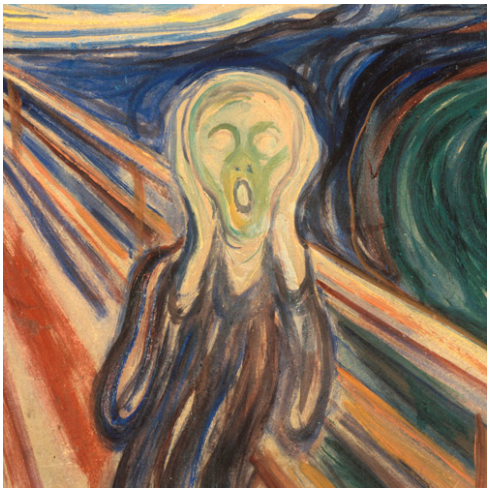


Se quiere comprobar si la longitud del lado del cuadrado o el número de experimentos tiene alguna influencia sobre el resultado obtenido, y para hacer mejor uso de la CPU del sistema se pide desarrollar un programa concurrente que permita lanzar múltiples hilos con múltiples pruebas a la vez. Para ello, se pide que desarrolle una clase Monte_Carlo_Pi con la siguiente estructura:

- Como **atributos**, el valor del lado del cuadrado y el número de experimentos a realizar. Estos valores serán inicializados por el **constructor**. Es posible que necesite atributos adicionales.
- Un método **realizar_estimacion()**, que ejecuta la simulación de Monte Carlo descrita para obtener el valor de PI. Utilice la clase `uniform_real_distribution<> (a, b)` para obtener un generador de números aleatorios reales entre los valores a y b.
- El método **operator<<** que imprime los datos del experimento y el resultado por consola. Por ejemplo: "*1000 experimentos con lado 19. Aprox PI: 3.192*"

Haga una función **main** que cree al menos 10 objetos de tipo Monte_Carlo_Pi con 1000 simulaciones y longitud de lado variable, y 10 objetos con longitud de lado 2 y número de simulaciones variable. Cree posteriormente 20 hilos, que ejecutarán el método realizar_estimacion(). Por último, imprima por consola los resultados. Solo puede utilizar la funcionalidad que proporciona la librería estándar de C++.

Las soluciones vienen a continuación.
Consultar solo si ya está un poco desesperad@, o para comparar soluciones



Puede que las soluciones sean un poco más generales o haya más detalles de los que se piden en el enunciado del ejercicio. Se proporcionan para que el alumno tenga un código más completo. Como sabe, en programación **NO** existe la solución única a un ejercicio.

EJERCICIO 1

```
#include <iostream>
#include <cmath>

using namespace std;

struct I_Forma {
    virtual double calcular_perimetro() const = 0;
    virtual double calcular_area() const = 0;
    friend ostream& operator<< (ostream &os, const I_Forma &f);
    I_Forma (int id) : id_{id} {}
    int get_id() const { return id_; }
protected :
    virtual void print(ostream &os) const = 0;
private:
    int id_;
};

ostream& operator<< (ostream &os, const I_Forma &f) {
    f.print(os);
    return os;
}

class Triangulo : public I_Forma {
    double lado1_, lado2_, lado3_;
protected:
    void print(ostream &os) const override {
        os << "Triangulo #"<<get_id()<<", l1="<<lado1_<<",
l2="<<lado2_<<", l3="<<lado3_;
    }
public:
    Triangulo (int id, double l1, double l2, double l3) : I_Forma{id},
lado1_{l1}, lado2_{l2}, lado3_{l3} {}

    double calcular_perimetro() const override {
        return lado1_+lado2_+lado3_;
    }

    double calcular_area() const override {
        double s = calcular_perimetro()/2.0;
        return sqrt(s*(s-lado1_)*(s-lado2_)*(s-lado3_));
    }

};

class Rectangulo : public I_Forma {
    double lado1_, lado2_;
```

```
protected:
    void print(ostream &os) const override {
        os << "Rectangulo #"<<get_id()<<", l1="<<lado1_<<", l2="<<lado2_;
    }
public:
    Rectangulo (int id, double l1, double l2) : I_Forma{id}, lado1_{l1},
lado2_{l2} {}

    double calcular_perimetro() const override {
        return (lado1_+lado2_)*2;
    }

    double calcular_area() const override {
        return (lado1_*lado2_);
    }

    double get_lado1 () const { return lado1_; }

};

class Cuadrado : public Rectangulo {
protected:
    void print(ostream &os) const override {
        os << "Cuadrado #"<<get_id()<<" de lado="<<get_lado1();
    }
public:
    Cuadrado (int id, double l) : Rectangulo{id, l, l} {}
};

void print_polimorfismo (I_Forma &i) {
    cout << i << ", area: "<<i.calcular_area()<<", perimetro:
"<<i.calcular_perimetro()<<'\n';
}

int main_ej2() {
    Triangulo t {1, 3, 4, 6};
    cout << t << ", area: "<<t.calcular_area()<<", perimetro:
"<<t.calcular_perimetro()<<'\n';
    Rectangulo r {2, 3, 4};
    cout << r << ", area: "<<r.calcular_area()<<", perimetro:
"<<r.calcular_perimetro()<<'\n';
    Cuadrado c {3, 3};
    cout << c << ", area: "<<c.calcular_area()<<", perimetro:
"<<c.calcular_perimetro()<<'\n';
    cout << "_________\n";
    print_polimorfismo(t);
    print_polimorfismo(r);
    print_polimorfismo(c);

    return 0;
}
```

EJERCICIO 2

```
#include <iostream>
#include <map>
#include <set>
#include <vector>

using namespace std;

struct Nodo {
    const int id_;
    Nodo (int id) : id_{id} {}
    int get_id () const { return id_; }
};

class Grafo {
    vector<Nodo> nodos_;
    map<Nodo*, set<Nodo*>> arcos_;
    int num_arcos_;
public:
    Grafo () : num_arcos_{0} {}

    bool add_nodo (const Nodo n) {
        for (const auto &i : nodos_) {
            if (i.get_id() == n.get_id()) { return false; }
        }
        nodos_.push_back(n);
        return true;
    }

    bool add_arco (const int n1, const int n2) {
        if (n1==n2) return false;
        Nodo *N1=nullptr, *N2=nullptr;
        for (auto &i : nodos_) {
            if (i.get_id() == n1) { N1=&i; }
            if (i.get_id() == n2) { N2=&i; }
        }
        if (!N1 or !N2) return false;
        const auto &[i,b] = arcos_[N1].insert(N2);
        if (b) {
            ++num_arcos_;
            arcos_[N2].insert(N1);
        }
        return b;
    }

    friend ostream& operator<< (ostream &os, const Grafo &g) {
        cout << "Grafo con " << g.nodos_.size() << " nodos y " <<
g.num_arcos_ << " arcos";
        for (const auto &[k,v] : g.arcos_) {
            cout << "\nNodo " << k->get_id()<<'\n';
            for (const auto &n : v) {
                cout << '\t' << n->get_id();
```

```
            }
        }
        return os;
    }

    int max_num_arcos_nodo () const {
        int tmp=0;
        for (const auto &[k,v] : arcos_) {
            if (tmp<v.size()) { tmp=v.size();  }
        }
        return tmp;
    }
};

int main() {
    Nodo n1 {1}, n2 {2}, n3{3}, n4{4}, n5{5}, n6{6};
    Grafo g;
    cout << boolalpha;
    cout << "1 (True): " << g.add_nodo(n1) << '\n';
    cout << "2 (False): " << g.add_nodo(n1) << '\n';
    cout << "3 (True): " << g.add_nodo(n2) << '\n';
    cout << "4 (False): " << g.add_arco(1,3) << '\n';
    g.add_nodo(n3);
    g.add_nodo(n4);
    g.add_nodo(n5);
    g.add_nodo(n6);


    cout << "5 (True): " << g.add_arco(1,2) << '\n';
    cout << "6 (False): " << g.add_arco(1,2) << '\n';
    g.add_arco(1,3);
    g.add_arco(1,4);
    g.add_arco(2,3);
    g.add_arco(2,4);
    g.add_arco(4,5);
    g.add_arco(1,6);
    cout << g << '\n';
    cout << "Max arcos de un nodo (4): "<< g.max_num_arcos_nodo() <<
'\n';
    return 0;
}
```

EJERCICIO 3: CONCURRENCIA

```
#include <thread>
#include <iostream>
#include <random>
#include <vector>

using namespace std;

std::random_device rd;
std::default_random_engine gen(rd());

class Monte_Carlo {
    const int lado_;
    const int n_exp_;
    double estimacion_;
    bool ha_terminado_;
public:
    Monte_Carlo (int lado, int experimentos) :  lado_{lado}, n_exp_
{experimentos}, ha_terminado_{false}  {}

    void realizar_estimacion () {
        int n_en_circulo=0;
        uniform_real_distribution<> aleatorio (0, lado_/2.0);
        for (int i=0; i<n_exp_; ++i) {
            double x=aleatorio(gen), y=aleatorio(gen);
            double distancia = hypot(x,y);
            if (distancia<lado_/2.0) ++n_en_circulo;
        }
        estimacion_ = 4.0*n_en_circulo/n_exp_;
        ha_terminado_ = true;
    }

    double get_estimacion() {
        if (ha_terminado_) return estimacion_;
        return 0.0;
    }

    friend ostream& operator<< (ostream &os, Monte_Carlo &m) {
        os <<m.n_exp_<< " experimentos con lado "<<m.lado_;
        if (m.ha_terminado_){
            os <<". Aprox PI: "<<m.estimacion_;
        }
        return os;
    }
};



int main_ej_concu() {
    vector<Monte_Carlo> estimaciones;
    estimaciones.reserve (100);
    for (int i=1; i<50; i+=2)
```

```
    {  estimaciones.emplace_back(i, 1000);    }

    for (int i=1; i<50; i+=2)
    {        estimaciones.emplace_back(10, i*1000);    }

    vector<thread> hilos;
    for (int i=0; i<estimaciones.size(); ++i)
    {hilos.emplace_back(&Monte_Carlo::realizar_estimacion,
&estimaciones[i]);}

    for (auto &h : hilos)
    {h.join();}
    for ( auto &e : estimaciones)
    {cout << e << '\n';}

    return 0;
}
```