

# TODOS-los-EJERCICIOS-de-POLIMORF...



**JesusLagares**



**Programación Orientada a Objetos**



**2º Grado en Ingeniería Informática**



**Escuela Superior de Ingeniería  
Universidad de Cádiz**

quieres trabajar en Wuolah???

tú puedes ayudarnos a llevar **WUOLAH**  
al siguiente nivel (o alguien que conozcas)

**TE  
BUSCAMOS**



sin ánimo de lucro, chequea esto:

No podemos asegurarte que le gustes  
a tu crush. Todo lo demás, sí.

Cleverea, seguros sinceros y sin peros.

# EJERCICIOS RESUELTOS

## POLIMORFISMO

¡Hola! ¿Me echabas de menos? Ya casi he terminado la carrera; no obstante, aún tengo que aprobar esta asignatura... Por eso me he dedicado a diseccionarla al 100% para entender perfectamente qué tengo que saber para poder aprobarla, de ahí la creación de este documento. En él encontrarás todos los ejercicios sobre polimorfismo que he podido encontrar completamente resueltos y explicados. No obstante, tengo que decirte que soy un alumno de informática como tú, por lo que, quizás en estos ejercicios haya algún error o encuentres algo que se puede explicar mejor. En ambos casos, te animo a que me contactes para poder mejorar estos apuntes. Nada asegura que apruebes con ellos, pero he dado mi mejor esfuerzo para que así sea. Ah, y no lo olvides 😊

MUCHA SUERTE Y A FULL HD QUE SE APRUEBA CARAJO

### Febrero 2021

#### Ejercicio 3 febrero 2021 (1,5 puntos)

Considera el siguiente programa:

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

class Instrumento {
public:
    Instrumento(string n): nom(n) {}
    void tocar() const {
        cout << "Soy un " << nom << " y pertenezco a "
            << tipo() << endl;
    }
    string nombre() const { return nom; }
    string tipo() const { return "instrumento"; }
private:
    string nom;
};

int main() {
    Instrumento *pi = new Cuerda("violin");
    pi->tocar();
    delete pi;
}
```

Defina una clase llamada *cuerda* de tal forma que el programa anterior se compile sin errores y al ejecutarlo imprima el texto "Soy un violin y pertenezco a cuerda". Si lo cree necesario, modifique la clase *Instrumento* para que en tiempo de ejecución se enlace con los métodos apropiados y no se produzca ningún tipo de problema.

Instrucciones de entrega:

- Escriba su respuesta a mano en papel.
- Coloque su DNI o tarjeta universitaria sobre cada página escrita (sin que tape el texto) y capture una imagen nítida.
- Renombré el fichero de cada imagen con su nombre y apellidos y añada un número correlativo de página.
- Suba cada fichero de imagen.

```
/*
A priori, este ejercicio podría resolverse de diferentes
maneras utilizando los conceptos relacionados con Polimorfismo;
no obstante, nos dicen que, si lo creemos necesarios, modifiquemos
la clase, pero no nos hablan de modificar el ejemplo de ejecución.
De la misma manera, nos dicen "para que en tiempo de ejecución
se enlace con los métodos apropiados". Estas son las 2 claves que
determinarán la forma exacta de realizar el ejercicio => En este
caso habrá que hacer que tipo() sea virtual e
implementarlos en una subclase Cuerda(), de este modo al llamarlos
desde un objeto Cuerda devolverá "cuerda"
*/
```

```

#include <iostream>
#include <string>
using namespace std ;

class Instrumento
{
public :
    Instrumento(string n): nom(n) {}
    void tocar() const
    {
        cout << "Soy un " << nombre() << " y pertenezco a " << tipo() << endl ;
    }
    string nombre() const
    {
        return nom ;
    }
    virtual string tipo() const
    {
        return "instrumento" ;
    }
private :
    string nom ;
} ;

```

```

class Cuerda: public Instrumento
{
public :
    Cuerda(string n): Instrumento(n) {}
    string tipo() const
    {
        return "cuerda" ;
    }
private :
} ;

```

```

int main ()
{
    Instrumento *pi = new Cuerda("violin") ;
    pi->tocar() ;
    delete pi ;
}

```

C:\Users\Zegar\Downloads>g++ -o main main.cpp  
C:\Users\Zegar\Downloads>main.exe  
Soy un violin y pertenezco a cuerda

## Junio 2020

Queremos construir la clase genérica ColaPrioridad<T> utilizando para ello el contenedor estándar para listas (List<T>). En una cola con prioridad siempre se extrae el elemento más prioritario. Por simplificar, consideraremos que la prioridad de un elemento viene dada por su propio valor y el orden por el operador < para el tipo de elementos.

Se quiere implementar la clase genérica ColaPrioridad<T> de elementos de tipo T, con los siguientes métodos:

- vacia(): comprueba el estado de la cola
- primero(): devuelve el elemento prioritario
- encolar(): añade un elemento a la cola
- desencolar(): elimina el elemento prioritario

¿Sería más conveniente utilizar composición o especialización? Seleccione la opción que considere más adecuada y explique el por qué. A continuación, implemente la clase genérica ColaPrioridad<T>.

Nota: Se recuerda que el contenedor estándar para listas posee funciones miembros como begin, end, empty, front, insert, erase etc...

```

/*
    Sería más conveniente utilizar la composición, debido a que una
    ColaPrioridad no es un subtipo del contenedor list (no podemos
    utilizar una ColaPrioridad en todos los casos que podamos utilizar
    un list). A su vez, con los métodos que nos dicen que hay que
    implementar, si lo modelamos como una especialización, ColaPrioridad
    heredaría métodos de List que después no utilizaría, como por ejemplo
    remove. Por este motivo, es mejor modelarlo como una composición.
*/
template <typename T>
class ColaPrioridad
{
public :
    bool vacia() const ;
    const T& primero() const ;
    void encolar();
    void desencolar(T& e) ;
private :
    List<T> colaPrioridad ; // Composición entre ColaPrioridad - list
}

```

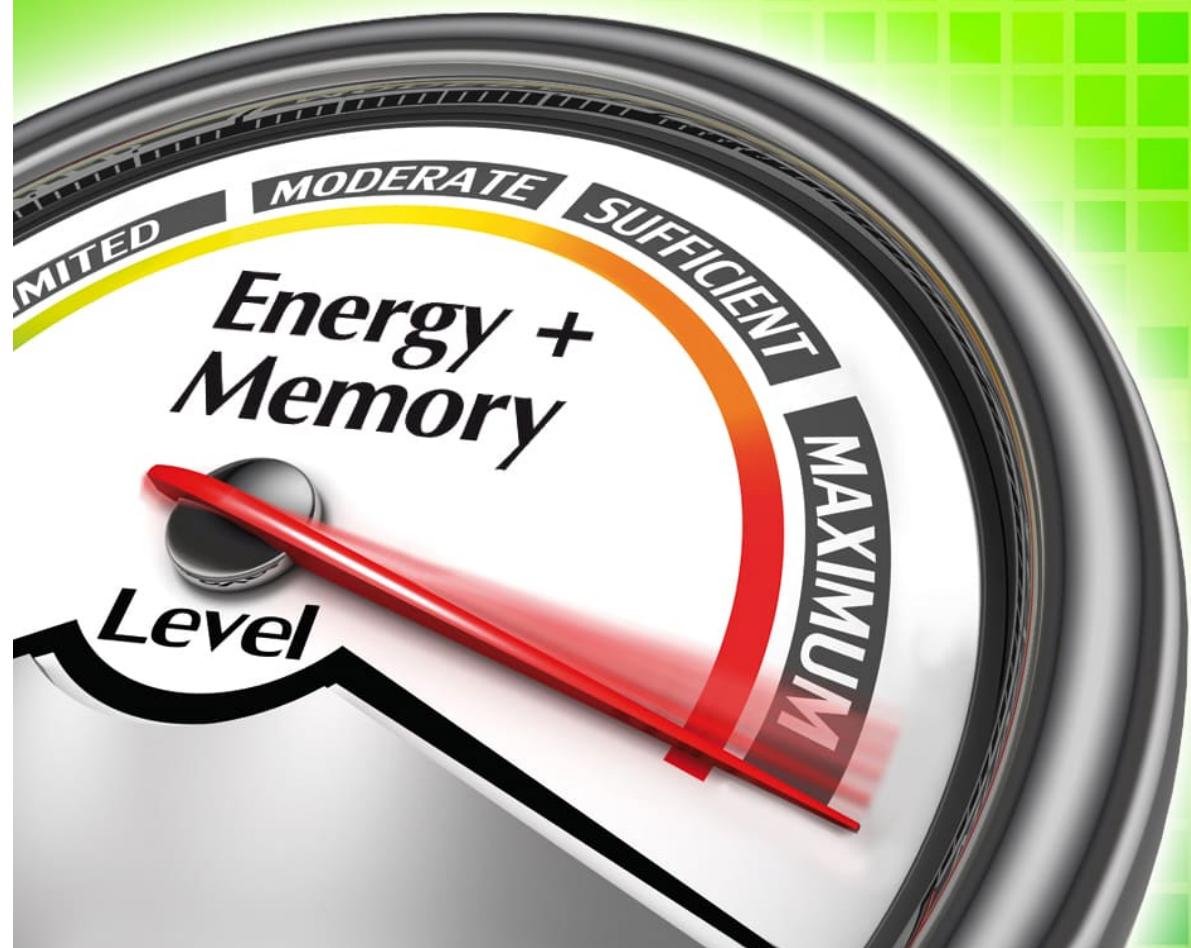
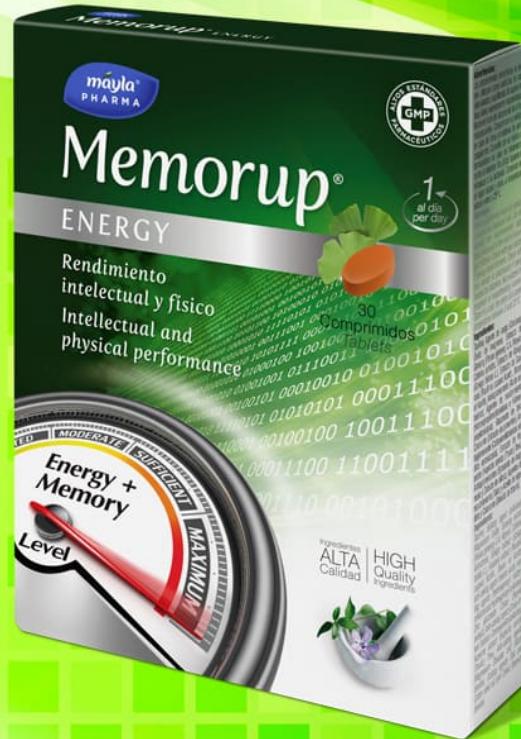
Jesús Lagares Galán

WUOLAH

# Memorup®

## ENERGY

- Rendimiento intelectual y físico
- Memoria, concentración y energía



## Junio 2020

```
#include <iostream>
#include <string>
#include <list>
using namespace std;

class Instrumento{
public:
    enum tclase{instrumento, percusión, cuerda, viento};
    Instrumento(string n): nom(n), clase_(instrumento){}
    void tocar()const{
        cout << "Soy un " << nombre() << " y pertenezco a " << clase() << endl;
    }
    string nombre()const{return nom;}
    string clase()const{
        switch(clase_){
            case percusión: return "percusión";
            case cuerda: return "cuerda";
            case viento: return "viento";
            default: return "instrumento";
        }
    }
protected:
    string nom;
    tclase clase_;
};

class Percusion: public Instrumento{
public:
    Percusion(string n): Instrumento(n){clase_=percusión;}
};

class Cuerda: public Instrumento{
public:
    Cuerda(string n): Instrumento(n){clase_=cuerda;}
};

class Viento: public Instrumento{
public:
    Viento(string n): Instrumento(n){clase_=viento;}
};

int main(){
    list<Instrumento*> l{new Percusion("tambor"),new Percusion("triangulo"),new Instrumento("sintetizador"), new Cuerda("piano")};
    for (Instrumento p: l){
        p->tocar();
        delete p;
    }
}
```

a) Escriba lo que el programa enviará a la salida estándar  
b) ¿Se puede mejorar la implementación de esta jerarquía de clases usando métodos polimórficos? Razona la respuesta, en caso afirmativo, reescriba las clases de tal forma que el resultado del programa sea idéntico.

```
#include <iostream>
#include <string>
#include <list>
using namespace std;

/*
a) El programa escribirá:
"Soy un tambor y pertenezco a percusión"
"Soy un triángulo y pertenezco a percusión"
"Soy un sintetizador y pertenezco a instrumento"
"Soy un piano y pertenezco a cuerda"

b) Sí, podemos mejorar la implementación haciendo que el método clase() se comporte como un método polimórfico gracias a la sobrecarga.
De esta manera podríamos reducir la repetición del código y la complejidad ciclomática generada por la instrucción switch
*/
class Instrumento {
public :
    Instrumento(string n): nom(n){}
    void tocar() const
    {
        cout << "Soy un " << nombre() << " y pertenezco a " << clase() << endl;
    }

    string nombre() const
    {
        return nom ;
    }

    virtual string clase() const // Hacemos que se comporte de forma polimórfica
    {
        return "Instrumento" ;
    } // Método virtual

private :
    string nom ;
};

class Percusion: public Instrumento
{
public :
    Percusion(string n):Instrumento(n) {}
    string clase() const
    {
        return "Percusión" ;
    } // Método virtual
};

class Cuerda: public Instrumento
{
public :
    Cuerda(string n):Instrumento(n) {};
    string clase() const
    {
        return "Cuerda" ;
    } // Método virtual
};

class Viento: public Instrumento
{
public :
    Viento(string n):Instrumento(n) {};
    string clase() const
    {
        return "Viento" ;
    } // Método virtual
};

int main ()
{
    list<Instrumento*> l{new Percusion("tambor"),new Percusion("triangulo"),new Instrumento("sintetizador"), new Cuerda("piano")};

    for (Instrumento *p: l){
        p->tocar();
        delete p;
    }

    return 0 ;
}
```

C:\Users\Zegan\Downloads>g++ -o main main.cpp  
C:\Users\Zegan\Downloads>main.exe  
Soy un tambor y pertenezco a Percusión|n  
Soy un triángulo y pertenezco a Percusión|n  
Soy un sintetizador y pertenezco a Instrumento  
Soy un piano y pertenezco a Cuerda

## Junio 2020

Necesitamos una jerarquía de clases para dos tipos de vehículos, turismos y camiones. De ambos se guardará la fecha de fabricación(string), mientras que los turismos tendrán, además, un número máximo de pasajeros y los camiones un peso máximo autorizado. Se incluirá un método para imprimir todos los datos de un vehículo y además, en todas las clases se dispondrá de un método observador para cada atributo.

Diseña una jerarquía de clases fácilmente extensible y de forma que todos los vehículos tengan que ser forzosamente de algún tipo turismo, camión o cualquier otro (motocicleta, furgoneta...etc) que se quiera añadir más adelante.

- Implemente la jerarquía de clases según los requerimientos descritos. (1)
- Escriba una función que clasifique un vector de punteros a vehículos en otros dos, uno de punteros a turismos y el segundo de punteros a camiones.(1)

### Apartado a)

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

/*
Utilizaremos herencia pública para conseguir que la jerarquía sea fácilmente extensible.
Además, al decirnos que todos los vehículos deben ser forzosamente algún tipo de turismo, camión, o cualquier otro, nos
hablan de que, NOAH es simplemente un Vehículo => No se podrán crear objetos de dicha clase, sino que todos los objetos
de la jerarquía serán de algún subtipo => Tenemos que crear una clase abstracta utilizando un método virtual puro
*/
class Vehiculo
{
public:
    Vehiculo(string fechaFabricacion):fechaFabricacion_(fechaFabricacion) {}
    string getFechaFabricacion() const // Get para la fecha de fabricación
    {
        return fechaFabricacion_;
    }
    virtual void imprimirDatosVehiculos() = 0; // Método que hace la clase abstracta

private:
    string fechaFabricacion_;
};

class Turismo: public Vehiculo
{
public:
    Turismo(string fechaFabricacion , int numeroMaximoPasajeros):Vehiculo(fechaFabricacion),numeroMaximoPasajeros_(numeroMaximoPasajeros) {}
    int getNumeroMaximoPasajeros() const // Observador para el número máximo de pasajeros
    {
        return numeroMaximoPasajeros_;
    }
    void imprimirDatosVehiculos()
    {
        cout << "El turismo se fabricó en el: " << this->getFechaFabricacion() << " y tiene capacidad para " << this->getNumeroMaximoPasajeros() << endl ;
    }
private:
    int numeroMaximoPasajeros_;
};

class Camion: public Vehiculo
{
public:
    Camion(string fechaFabricacion , int pesoMaximoAutorizado):Vehiculo(fechaFabricacion),pesoMaximoAutorizado_(pesoMaximoAutorizado) {}
    int getPesoMaximoAutorizado() const
    {
        return pesoMaximoAutorizado_;
    }
    void imprimirDatosVehiculos()
    {
        cout << "El camión se fabricó en el: " << this->getFechaFabricacion() << " y tiene un peso máximo autorizado de " << this->getPesoMaximoAutorizado() << endl ;
    }
private:
    int pesoMaximoAutorizado_;
};

int main ()
{
    Turismo t("28 de Julio", 5);
    Camion c("27 de Marzo" , 500);

    t.imprimirDatosVehiculos();
    c.imprimirDatosVehiculos();

    return 0;
}
```

## No podemos asegurarte que le gustes a tu crush. Todo lo demás, sí.

Cleverea, seguros sinceros y sin peros.

### Apartado b)

```
void clasificarVectorVehiculos(vector<Vehiculo*> vectorVehiculo)
{
    // Creamos los vectores para la clasificación
    vector<Turismo*> vectorTurismo ;
    auto iteradorVectorTurismo = vectorTurismo.begin() ;

    vector<Camion*> vectorCamion ;
    auto iteradorVectorCamion = vectorCamion.begin() ;

    int contadorAuxiliar = 0 ;

    for ( auto iterador = vectorVehiculo.begin() ; iterador != vectorVehiculo.end() ; iterador++ , contadorAuxiliar++ )
    {
        // Detectamos el tipo con dynamic_cast (para hacer conversiones "hacia abajo")
        if ( Turismo* turismoAuxiliar = dynamic_cast<Turismo*>(vectorVehiculo[contadorAuxiliar]) )
        {

            vectorTurismo.insert( iteradorVectorTurismo , vectorVehiculo[contadorAuxiliar] ) ;
            vectorVehiculo.erase( iterador ) ; // Eliminamos el elemento
            iteradorVectorTurismo++ ;

        }

        else // Asumimos que, si no es del tipo de Turismo será de tipo Camion (si tienes dudas sobre esto pregunta en el examen)
        {

            vectorTurismo.insert( iteradorVectorCamion , vectorVehiculo[contadorAuxiliar] ) ;
            vectorVehiculo.erase( iterador ) ; // Eliminamos el elemento
            iteradorVectorCamion++ ;

        }
    }
}
```

## Junio 2020

Sea `Instrumento` una clase abstracta de las que derivan las clases `Bombo`, `Guitarra`, `Clarinete` y otras. Supongamos que existe una función para tocar instrumentos definida como sigue:

```
#include <typeinfo>

void tocar(Instrumento& i)
{
    if(typeid(i)==typeid(Bombo))
    {
        //hacer sonar el bombo
        cout << "tocar bombo" << endl;
    }
    else if(typeid(i)==typeid(Guitarra)){
        //hacer sonar la guitarra
        cout << "tocar guitarra" << endl;
    }
    else if(typeid(i)==typeid(Clarinete)){
        //hacer sonar el clarinete
        cout << "tocar el clarinete" << endl;
    }
    //y otras
}
```

- Escriba unas líneas de código, como ejemplo de uso de la función `tocar()`, que permitan apreciar su comportamiento polimórfico con diversos tipos de instrumentos.
- ¿Cree que ésta es la mejor forma de implementar la acción de hacer sonar los diferentes instrumentos? Razone la respuesta. En caso negativo, describa cómo mejorarlala, escriba todo el código necesario y modifique el ejemplo anterior para mostrar la diferencia de uso.

Apartado a)

```
int main ()
{
    Bombo b ;
    Guitarra g ;
    Clarinete c ;

    tocar(b) ;
    tocar(g) ;
    tocar(c) ;
}
```

Apartado b)

```
class Bombo: public Instrumento
```

```
class Guitarra: public Instrumento
```

```
class Clarinete: public Instrumento
```

```
int main ()
```

```
Bombo b ;  
Guitarra g ;  
Clarinete c ;  
  
b.tocar() ;  
g.tocar() ;  
c.tocar() ;
```

۲

## Diciembre 2020

Sea **Figura** una clase abstracta de la que derivan las clases **Círculo**, **Triángulo**, **Cuadrado** y otras. Supongamos que existe una función para rotar figuras definida como sigue:

```
#include <typeinfo>

void rotar(Figura& fig)

{

    if (typeid(fig) == typeid(Círculo)) {

        // no hacer nada

    }

    else if (typeid(fig) == typeid(Triángulo)) {

        // rotar triángulo

    }

    else if (typeid(fig) == typeid(Cuadrado)) {

        // rotar cuadrado

    }

    // y otras

}
```

a) Ponga un ejemplo de uso de la función `rotar()` que permita apreciar su comportamiento polimórfico. (0,5 p)

b) ¿Cree que ésta es la mejor forma de implementar la rotación de figuras? Razone la respuesta. En caso negativo, describa cómo mejorarla y emplee el ejemplo anterior para mostrar la diferencia de uso. (1,0 p)

Apartado a)

```
int main ()
{
    Círculo c ;
    Triángulo t ;
    Cuadrado cu ;

    rotar(c) ;
    rotar(t) ;
    rotar(cu) ;

}
```

# No podemos asegurarte que le gustes a tu crush. Todo lo demás, sí.

Cleverea, seguros sinceros y sin peros.

## Apartado b)

```
#include <iostream>
#include <typeinfo>

using namespace std ;

/*
b) No, ya que de esta manera estamos generando un aumento
de la complejidad ciclomática y estamos repitiendo código
a cada figura que se añade dentro de la jerarquía de clases.
Además, rotar una Figura es un método propio de la clase,
por lo que sería más lógico modelar el método como una
operación de la misma. Por esto, para mejorarlo,
definiría en Figura un método virtual puro rotar() y lo
implementaría de diferente forma a lo largo de la jerarquía
según cómo se rote cada figura. De esta manera convertiría Figura
en una clase abstracta, ya que en el ejemplo no aparece que
se cree ningún objeto de la misma.

*/

class Figura
{
public :
    virtual void rotar() = 0 ; // Método virtual puro
private :
};

class Circulo: public Figura
{
public :
    void rotar()
    {
        cout << "rotar circulo" << endl ;
    }
private :
};

class Triangulo: public Figura
{
public :
    void rotar()
    {
        cout << "rotar triangulo" << endl ;
    }
private :
};

class Cuadrado: public Figura
{
public :
    void rotar()
    {
        cout << "rotar cuadrado" << endl ;
    }
private :
};

int main ()
{
    Circulo c ;
    Triangulo t ;
    Cuadrado cu ;

    c.rotar() ;
    t.rotar() ;
    cu.rotar() ;
}
```

## Examen 2015

5. Hay que hacer una biblioteca que trabaje con formas geométricas simples bidimensionales; de momento, nos interesa de ellas el área. Escriba todo lo necesario para que el siguiente programa funcione correctamente. En concreto, deberá escribir apropiadamente las clases *Forma*, *Círculo* y *Cuadrado*.

3,0 p  
1,5 p

```
int main()
{
    Círculo cir(1.0);
    Cuadrado cua(2.0);
    Formas* vpf[] = { &cir, &cua };
    forma
    for (int i = 0; i < 2; i++)
        cout << "El área de la figura " << (i + 1) << " es " << forma->area();
}
```

La salida del programa anterior deberá ser:

```
El área de la figura 1 es 3.14159
El área de la figura 2 es 4
```

Ahora modifique la clase *Círculo* para que el constructor lance la excepción *Radio\_negativo* cuando el radio suministrado como argumento sea negativo, evidentemente. *Radio\_negativo* será pues una clase dentro de *Círculo*, que tendrá un atributo *rd* que será el radio inválido (negativo), un constructor, y un método constante observador llamado *valor* que nos da el valor del atributo.

0,5 p

Escriba también un programa de ejemplo dónde haga uso de esta excepción.

1,0 p

### Apartado a)

```
#include <iostream>
#include <typeinfo>
#include <string>
using namespace std ;

class Forma
{
public :
    Forma(double tamano):tamano_(tamano) {}
    virtual double area() = 0 ;
private :
    double tamano_ ;
} ;

class Círculo: public Forma
{
public :
    Círculo(double tamano):Forma(tamano) {}
    double area()
    {
        return 3.14 ;
    }
private :
} ;

class Cuadrado: public Forma
{
public :
    Cuadrado(double tamano):Forma(tamano) {}
    double area()
    {
        return 4;
    }
private :
} ;

int main ()
{
    Círculo cir(1.0) ;
    Cuadrado cua(2.0) ;
    Forma* forma[] = {&cir , &cua} ;

    for (int i = 0 ; i < 2 ; i++)
    {
        cout << "El área de la figura " << (i+1) << " es " << forma[i]->area() << endl;
    }
    return 0 ;
}
```

## Apartado b)

```
#include <iostream>
#include <typeinfo>
#include <string>
using namespace std ;

class Forma
{
public :
    Forma(double tamano):tamano_(tamano) {}
    virtual double area() = 0 ;
private :
    double tamano_ ;
};

class Circulo: public Forma
{
public :
    Circulo(double tamano):Forma(tamano){
        if ( tamano < 0 )
        {
            throw Radio_Negativo(tamano) ; // Lanzamos la excepcion
        }
    }
    double area()
    {
        return 3.14 ;
    }
};

class Radio_Negativo // Clase excepcion
{
public :
    Radio_Negativo(double rd):rd_(rd) {}
    double getRadioNegativo() const
    {
        return rd_ ;
    }
private :
    double rd_ ;
};

class Cuadrado: public Forma
{
public :
    Cuadrado(double tamano):Forma(tamano) {}
    double area()
    {
        return 4;
    }
private :
};

int main ()
{
    try
    {
        Circulo c(-1.0) ;
    }
    catch (const Circulo::Radio_Negativo& radioNegativo)
    {
        cout << "Introducido radio negativo: " << radioNegativo.getRadioNegativo() ;
    }
    return 0 ;
}
```

# Septiembre 2008

## Examen 08 ejercicio

martes, 23 de noviembre de 2021 8:33

4. Sea *Figura* una clase abstracta de la que derivan las clases *Circulo*, *Triangulo*, *Cuadrado* y otras. Supongamos que existe una función para rotar figuras definida como sigue:

```
void rotar(const Figura& fig)
{
    if (typeid(fig) == typeid(Circulo)) {
        // no hacer nada
    }
    else if (typeid(fig) == typeid(Triangulo)) {
        // rotar triángulo
    }
    else if (typeid(fig) == typeid(Cuadrado)) {
        // rotar cuadrado
    }
    // y otras
}
```

- Ponga un ejemplo de uso de la función *rotar*.
- ¿Cree que ésta es la mejor forma de implementar la rotación de figuras? Razone la respuesta. En caso negativo, describa cómo mejorarla y emplee el ejemplo anterior para mostrar la diferencia de uso.

### Apartado a)

```
int main ()
{
    Circulo c ;
    Triangulo t ;
    Cuadrado cu ;

    rotar(c) ;
    rotar(t) ;
    rotar(cu) ;
}
```

# No podemos asegurarte que le gustes a tu crush. Todo lo demás, sí.

Cleverea, seguros sinceros y sin peros.

## Apartado b)

```
#include <iostream>
#include <typeinfo>

using namespace std ;

/*
    b) No, ya que de esta manera estamos generando un aumento
    de la complejidad ciclomática y estamos repitiendo código
    a cada figura que se añade dentro de la jerarquía de clases.
    Además, rotar una Figura es un método propio de la clase,
    por lo que sería más lógico modelar el método como una
    operación de la misma. Por esto, para mejorarlo,
    definiría en Figura un método virtual puro rotar() y lo
    implementaría de diferente forma a lo largo de la jerarquía
    según cómo se rote cada figura. De esta manera convertiría Figura
    en una clase abstracta, ya que en el ejemplo no aparece que
    se cree ningún objeto de la misma.

*/

class Figura
{
public :
    virtual void rotar() = 0 ; // Método virtual puro

private :
};

class Circulo: public Figura
{
public :
    void rotar()
    {
        cout << "rotar circulo" << endl ;
    }
private :
};

class Triangulo: public Figura
{
public :
    void rotar()
    {
        cout << "rotar triangulo" << endl ;
    }
private :
};

class Cuadrado: public Figura
{
public :
    void rotar()
    {
        cout << "rotar cuadrado" << endl ;
    }
private :
};

int main ()
{
    Circulo c ;
    Triangulo t ;
    Cuadrado cu ;

    c.rotar() ;
    t.rotar() ;
    cu.rotar() ;
}
```

## Cola Prioridad

Revisa bien en este ejercicio la utilización de los contenedores. A pesar de haberle dado una solución, y compilar sin errores, todavía tengo dudas sobre su correcta utilización.

### Ejercicio 2 (2,5 puntos)

En una cola con prioridad siempre se extrae el elemento más prioritario. Por simplificar, consideraremos que la prioridad de un elemento viene dada por su propio valor y el orden por el operador < para el tipo de los elementos.

Se quiere implementar la clase genérica `ColaPrioridad<T>` de elementos de tipo `T`, con los siguientes métodos:

- `vacia()`: comprobar el estado de la cola.
- `primero()`: devuelve el elemento prioritario.
- `encolar()`: añade un elemento a la cola.
- `desencolar()`: elimina el elemento prioritario.

Para ello se usará la clase genérica estándar `list` y se proponen dos opciones:

- Definir `ColaPrioridad` como una especialización de `list`.
- Definir `ColaPrioridad` como una composición de `list`.

Seleccione la opción que considere más adecuada y explique el porqué. A continuación implemente la clase genérica `ColaPrioridad<T>`.

**Nota:** Se recuerda que el contenedor estándar para listas posee funciones miembro como `begin`, `end`, `empty`, `front`, `insert`, `erase`, etc.

Instrucciones de entrega:

- Escriba a mano su respuesta en papel.
- Coloque su DNI o tarjeta universitaria sobre cada página escrita (sin que tape el texto) y capture una imagen nítida.
- Renombre el fichero de cada imagen con su nombre y apellidos y añada un número correlativo de página.
- Suba cada fichero de imagen.

**Tiempo máximo 35 minutos**

```
#include <iostream>
#include <list>
using namespace std ;
/*
    ¿Es realmente una ColaPrioridad un subtipo de List<T>? No, la verdad es que no, como mucho sería un subtipo de una clase base Cola.
    ¿Podemos utilizar una ColaPrioridad en todos los casos que utilicemos una List<T>? No, ya que con una ColaPrioridad, y los métodos que nos dicen que implementemos
    no podríamos utilizarlo en los mismos casos.
    Además, si lo modelásemos como una herencia, ColaPrioridad heredaría todos los métodos de List<T>, encontrándose entre estos algunos que no tendría sentido utilizar
    con una cola, y otros que, de hacer un mal uso de ellos harían que la ColaPrioridad se comportase de una forma incorrecta.
    Por este motivo, se propone modelar ColaPrioridad como una composición de List.

*/
template <typename T>
class ColaPrioridad
{
public :
    bool vacia() const ;
    const T& primero() const ;
    void encolar(const T& elementoAEncolar) ;
    void desencolar() ;

private :
    list<T> colaPrioridad ;
} ;
template <typename T>
const T& ColaPrioridad<T>::primero() const
{
    return colaPrioridad.front() ;
}

template <typename T>
void ColaPrioridad<T>::encolar(const T& elementoAEncolar)
{
    colaPrioridad.insert( colaPrioridad.begin() , elementoAEncolar ) ;
    colaPrioridad.sort () ;// Habría que preguntarlo en el examen, ya que si sort (ordena de forma ascendente) no ordenase como queremos, habría que
    // recorrer toda la cola comparando los elementos según la prioridad e insertar donde toque
}

template <typename T>
void ColaPrioridad<T>::desencolar()
[ ]
    colaPrioridad.erase( colaPrioridad.begin() ) ;
]
```

## Elipse Circunferencia

Una circunferencia se puede considerar una elipse -----, donde sus dos radios rx y ry son iguales. Partiendo de esta consideración, el objetivo es definir dos clases Elipse y Circunferencia, utilizando la primera para definir la segunda.

La clase Elipse, dispondrá de dos métodos observador, radio\_x() y radio\_y(), mientras que la clase Circunferencia sólo tendrá un observador llamado radio(). En ambas clases existirá un método escala() que recibirá un factor de escala y modificará los radios en dicho factor. Por último, la clase Elipse tendrá sobrecargado el método escala() para recibir dos factores independientes, uno para escalar rx y otro para ry.

a) Escriba todo lo necesario para implementar ambas clases, justificando razonadamente la opción elegida como más conveniente entre las dos siguientes:

- Circunferencia se define por composición de una Elipse cuyos radios son iguales y sus métodos se implementan mediante delegación en los de Elipse.
- Circunferencia es una clase derivada de Elipse.

b) Defina una relación entre las clases anteriores y una nueva llamada FiguraPlana, con dos métodos:

- Escala(), para cambiar el tamaño de una figura en la misma proporción en las dos dimensiones.
- Y dibuja(), para dibujar la figura (tenga en cuenta que el procedimiento de dibujo será diferente para cada tipo de figura).

Describe la interfaz de las clases Elipse y Circunferencia si decide modificarlas.

### Apartado a)

```
/*
¿Podemos utilizar una circunferencia en todos los casos que utilicemos una elipse de radios iguales? Si.
¿Es una Circunferencia un subtipo de Elipse? Sí. De hecho en el enunciado nos dicen "Una circunferencia se puede considerar una elipse".
Además de esto, los métodos que nos dicen en el enunciado que hay que implementar para Elipse podemos utilizarlos en su totalidad para definir
el comportamiento de una circunferencia, por este motivo elijo modelarlo como una especialización.]
```

```
using namespace std;
class Elipse
{
public :
    Elipse(double rx, double ry):rx_(rx) , ry_(ry) {}

    double radio_x() const
    {
        return rx_ ;
    }

    double radio_y() const
    {
        return ry_ ;
    }

    void escala (const double factorRX , const double factorRY)
    {
        this->rx_ *= factorRX ;
        this->ry_ *= factorRY ;
    }

private :
    double rx_ , ry_ ;
};

class Circunferencia : public Elipse
{
public :
    Circunferencia(double r): Elipse(r, r) {}

    double radio () const
    {
        return this->radio_x() ; // Igual a cuál de los dos llamemos debido a que en la Circunferencia rx = ry
    }

    void escala (const double factor)
    {
        this->Elipse::escala(factor , factor) ; // Resolvemos el ámbito debido a que es una sobrecarga interna de métodos
    }

private :
};

int main ()
{
    Elipse e(2.0 , 3.0) ;
    cout << e.radio() << endl ;
    e.escala(4.0 , 5.0) ;
    cout << e.radio_x() << endl ;

    Circunferencia c(6.0) ;
    cout << c.radio() << endl ;
    c.escala(2.0) ;
    cout << c.radio() << endl ;

    return 0 ;
}
```

```
C:\Users\Zeger\Downloads>g++ -o main main.cpp
C:\Users\Zeger\Downloads>main.exe
2
3
6
12
```

Jesús Lagares Galán

WUOLAH

Tenemos lo que nos faltaba: Imprime tus apuntes al mejor precio y recíbelos en casa

## Apartado b)

```
Como en FiguraPlana podemos reutilizar el método dibuja() que utilizaba Ellipse, Circunferencia, y todas las figuras posteriores, la modifiqué como una superclase de FiguraPlana. Además, mantuve Ellipse como superclase de Circunferencia.

class FiguraPlana // Baso dimensiones
{
public:
    FiguraPlana(double rx_, double ry_):rx_(rx_), ry_(ry_) {}

    // Habría que preguntar en el examen, por ejemplo, yo he asumido que podía poner las getters en esta superclase, pero en el enunciado no nos dicen nada de ex
    double radio_x() const
    {
        return rx_;
    }

    double radio_y() const
    {
        return ry_;
    }

    void escala (const double factorRX , const double factorRY)
    {
        rx_*rx_ *= factorRX;
        ry_*ry_ *= factorRY;
    }

    virtual void dibuja() = 0; // Método virtual para dibujar cada figura en función, asumimos que de FiguraPlana no se pueden crear objetos
private:
    double rx_, ry_;
};

int main()
{
    Ellipse e(2.0, 1.0);
    e.dibuja();

    Circunferencia c(6.0);
    c.dibuja();
}

class Ellipse: public FiguraPlana
{
public:
    Ellipse(double rx, double ry):FiguraPlana(rx, ry) {}

    void dibuja()
    {
        cout << "Dibujando Ellipse" << endl;
    }

    private:
};

class Circunferencia: public Ellipse
{
public:
    Circunferencia(double r): Ellipse(r, r) {}

    double radio () const
    {
        return (this->radio_x());
    }

    void dibuja()
    {
        cout << "Dibujando Circunferencia" << endl;
    }

    private:
};

```

# No podemos asegurarte que le gustes a tu crush. Todo lo demás, sí.

Cleverea, seguros sinceros y sin peros.

## Clase Gato

1. Crea una nueva clase llamado Gato, y modifique el siguiente código de la clase Mamifero para que haga polimorfismo en tiempo de ejecución.

```
#include <iostream>
#include <string>
using namespace std;
class Mamifero{
public:
    Mamifero(string s) : nombre_(s) {}
    virtual void saludo() const{
        cout << "El nombre del animal es " << nombre() << " y es un " <<
        tipo() << endl;
    }
    virtual string tipo() const{ return "animal"; }
    string nombre() const{ return nombre_; }
private:
    string nombre_;
};

int main(){
    Mamifero* mp = new Gato("Miai");
    mp->saludo();
    delete mp;
}
```

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

class Mamifero
{
public :
    Mamifero(string s) : nombre_(s) {}
    virtual void saludo() const
    {
        cout << "El nombre del animal es " << nombre() << " y es un " << tipo() << endl ;
    }
    virtual string tipo() const
    {
        return "animal" ;
    }
    string nombre() const
    {
        return nombre_ ;
    }
private:
    string nombre_ ;
};
```

```
class Gato: public Mamifero
{
public :
    Gato(string s):Mamifero(s) {}
    string tipo() const
    {
        return "gato" ;
    }
private :
};

int main ()
{
    Mamifero* mp = new Gato("Miai");
    mp->saludo();
    delete mp;
}
```

## Clase Par

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

template <class T1, class T2>
class Par{
public:
    Par(): prime(T1()), segun(T2()) {}
    Par(const T1& p, const T2& s): prime(p), segun(s) {}
    T1 primero() const {return prime;}
    T1& primero() {return prime;}
    T2 segundo() const {return segun;}
    T2& segundo() {return segun;}
protected:
    T1 prime;
    T2 segun;
};
```

- a) Sobrecarga el operador << para la plantilla de clase par.
- b) Defina un tipo racional a partir de ella para representar números racionales.  
Sobrecargue el operador suma de números racionales.
- c) Defina una clase complejo con parte real e imaginaria de tipo doublé. Sobrecargue el operador de autosuma (+=) de un número complejo.

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

template <class T1, class T2>
class Par{
public:
    Par(): prime(T1()), segun(T2()) {}
    Par(const T1& p, const T2& s): prime(p), segun(s) {}
    T1 primero() const
    {
        return prime;
    }

    T1& primero()
    {
        return prime;
    }

    T2 segundo() const
    {
        return segun;
    }

    T2& segundo()
    {
        return segun;
    }
protected:
    T1 prime;
    T2 segun;
}

// Apartado a)
template <class T1, class T2>
ostream& operator <<(ostream& bufferDeFlujo , const Par<T1,T2>& par)
{
    bufferDeFlujo << "Primero: " << par.primero() << endl << "Segundo: " << par.segundo() << endl ;
    return flujo ,
}
```

```
// Apartado b)
template <class T1, class T2>
class Racional: public Par<int, int>
{
public :
    Racional(int numerador, int denominador):Par(numerador, denominador) {}

private :
}

Racional operator+ (const Racional& r1 , const Racional& r2)
{
    return Racional( r1.primero() * r2.segundo() + r2.primero() * r1.segundo() , r1.segundo() * r2.segundo() );
}
```

```
// Apartado c)
class Complejo: public Par<double, double>
{

public :
    Complejo(double real, double imaginaria):Par(real, imaginaria) {}

    Complejo operator +=(const Complejo& complejoDado)
    {
        primero += complejoDado.primero();
        segundo += complejoDado.segundo();

        return Complejo(primer(), segundo());
    }
private :
}
```

## Listado Ordenada

3) Se quiere implementar la clase *ListaOrdenada* de numero en coma flotante y doble precisión (posibles repetidos) con las operaciones que se declaran:

4 puntos

```
class ListaOrdenada
{
public:
    typedef list<double> const_iterator; //bidireccional
    listaOrdenada(); // Construir una lista vacía
    void insertar (double e); //Insertar elemento
    iterator buscar (doublé e) const; // Iterador al primer elemento e
    iterator begin( ) const; // Primera posición de la lista
    iterator end ( ) const; // Posición siguiente a la del último

private:
    //...
};
```

Para ello se usarán la clase genérica estándar *list* y se propone 2 opciones:

- Definir *ListaOrdenada* como una especialización de *list*.
- Definir *ListaOrdenada* como una composición de *list* y usar la técnica de delegación de operaciones.

a) Implemente *ListaOrdenada* seleccionando la opción que considere más adecuada y explique por qué.

3 puntos

b) Añada un nuevo método *size\_t contar (double e) const*, que cuenta el número de ocurrencias de un elemento dado. Utiliza para ello el algoritmo de la STL *count\_if()* el cual recibe dos iteradores y un predicado (clase objeto función cuyo operador () devuelve un bool). Defina el predicado como una función anónima (lambda).

1 punto

\*Nota: Se recuerda que el contenido estándar para listas posee funciones miembros *begin()*, *end()*, *cbegin()*, *cend()*, *insert()*, *erase()*, etc...

\*Nota\*: No contiene *find()*

### Apartado a)

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

/*
Podemos utilizar una ListaOrdenada en todos los momentos que utilicemos una lista? SI.
¿Es una ListaOrdenada un subtipo de Lista? SI.
Además de esto, ListaOrdenada puede hacer uso de todas las operaciones de list() sin ningún problema.
*/
class ListaOrdenada: public list<double>
{
public :
    typedef list<double>::const_iterator const_iterator;
    ListaOrdenada()
    {
        void insertar (double e);
        const_iterator buscar (double e) const ;
        const_iterator begin() const ;
        const_iterator end() const ;
    }
private :
    listaOrdenada::const_iterator ListaOrdenada::begin() const
    {
        return cbegin();
    }
    listaOrdenada::const_iterator ListaOrdenada::end() const
    {
        return cend();
    }
};

void ListaOrdenada::insertar(double e)
{
    auto iterador = list<double>::begin(); // Iterador a la primera posición
    while ( iterador != list<double>::end() && e > *iterador )
    {
        iterador++;
    }
    insert(iterador , e);
}

ListaOrdenada::const_iterator ListaOrdenada::buscar(double e) const
{
    const_iterator iterador = cbegin(); // Iterador a la primera posición
    while ( iterador != cend() && e != *iterador )
    {
        iterador++;
    }
    return iterador;
}
```

## Apartado b)

```
size_t ListaOrdenada::contar(double e) const
{
    return count_if( begin() , end() , [e](double val){return val == e;} );
/*
    Un predicado es un objeto función unario/binario (un tipo de objeto de la librería STL) que devuelve un valor lógico;
    es decir, una función booleana del tipo
        bool funcionBooleana()

    En este caso, utilizamos un mecanismo de C++ denominado Lambda Expressions (nos lo dicen en el enunciado) que nos permite definir objetos funciones (funciones)
    de forma anónima [], o encerrando entre esos corchetes el valor que queremos que se le pase a la función, además del argumento dado entre paréntesis.
    En este caso, como queremos encontrar el número de coincidencias iguales, necesitaremos comparar el valor que nos dan como argumento en contar() y el valor actual
    de cada posición de la ListaOrdenada.
    Por eso, hacer [e] sería crear una función anónima que recibe el valor e; pero además, le podemos pasar otro valor como argumento escribiendo entre paréntesis
    [e](double va)
y ya creamos el cuerpo de la función lógica
[e](double val)
{
    return val == e ;
}
Y así hemos creado un predicado (función lógica)
*/
}
```