Universidad de Jaén
Departamento de Informática

Programación Orientada a Objetos

http://bit.ly/poouja

Curso 2019/2020

José Ramón Balsas Almagro
Víctor Manuel Rivas Santos
Ángel Luis García Fernández
Juan Ruiz de Miras

Práctica 8. Polimorfismo

Objetivos

- Conocer el concepto de polimorfismo de objetos ligado a la herencia
- Practicar la resolución de ambigüedad en llamadas a métodos con objetos polimórficos.
- Comprender el concepto de polimorfismo estático o paramétrico y utilizarlo en C++ implementando y usando plantillas de funciones.
- Saber utilizar flujos en C++ para recuperar información de archivos de texto.

Índice

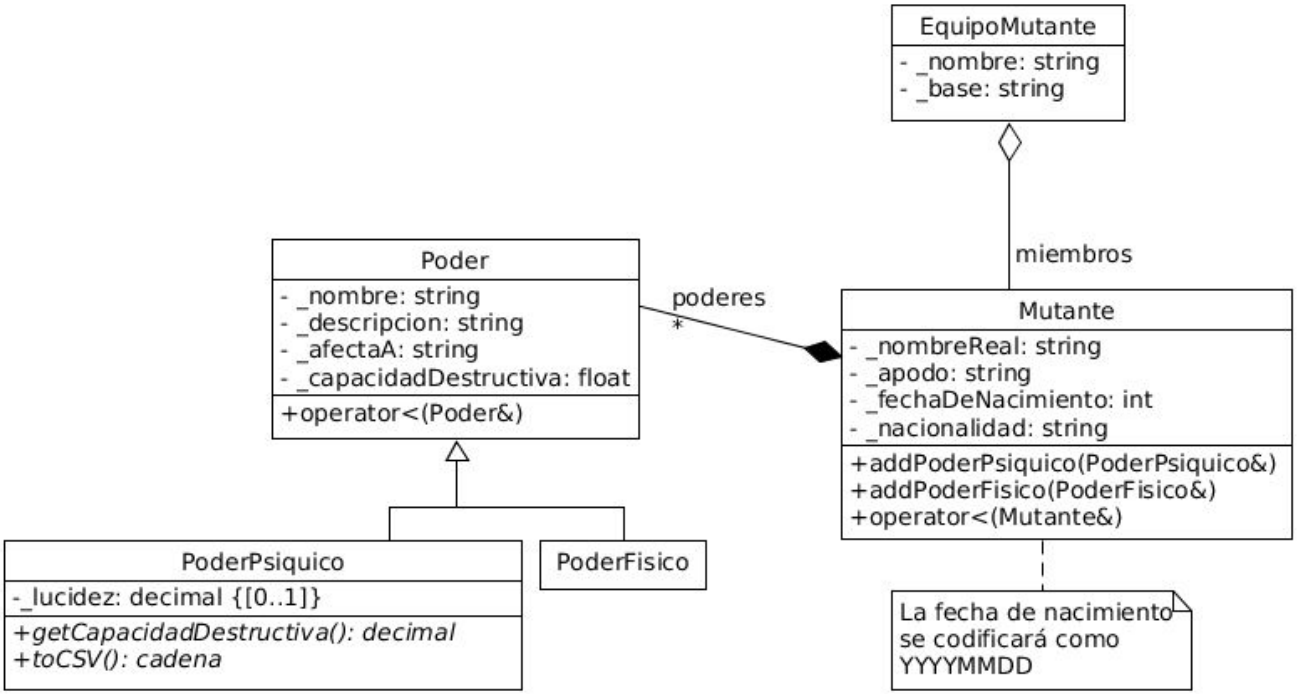
Contexto de la práctica

El equipo de desarrolladores de la organización S.H.I.E.L.D. ha observado que los Poderes de los mutantes tienen características específicas que afectan a su comportamiento dependiendo del tipo al que pertenezcan. En particular, han descubierto que los poderes psíquicos pueden verse afectados por un factor multiplicativo en el intervalo [0, 1], que han denominado *lucidez, y* que puede reducir considerablemente el daño máximo de este tipo de poderes.

Además, necesitan elaborar un ranking de mutantes que puedan calcular rápidamente, y que permita mostrar a todos los mutantes conocidos por su capacidad total destructiva, de forma que sea posible conocer quién es el más efectivo de todos ellos.

Por todos estos motivos, es necesario realizar las siguientes modificaciones en el diseño de la aplicación para atender a los nuevos cambios en los requisitos iniciales del sistema.

Afortunadamente, para los gestores de S.H.I.E.L.D. fue un acierto que su equipo de desarrollo cursara la asignatura de POO y planteara desde un principio un diseño orientado a objetos que facilitara enfrentarse a los terroríficos "requisitos mutantes", auténticos villanos del desarrollo software.



A partir del diseño inicial de la aplicación, se plantea un nuevo diagrama UML, y se hace necesario realizar los cambios propuestos. Como es costumbre para S.H.I.E.L.D., la amenaza terrorista es en los últimos meses más importante que nunca, y los programadores de S.H.I.E.L.D. no dan abasto con todo el trabajo pendiente, así que vuelven a solicitar tu ayuda incondicional (o sea, gratuita) para terminar la aplicación en un tiempo récord; incluso te han prometido que, de seguir así, en 5 o 10 años te podrían hacer un contrato en prácticas interinas no remuneradas con solo pagar una matrícula y una cuota mensual de afiliación al Sindicato Informático Mutante (SIM).

Punto de partida

El punto de partida de esta práctica son las clases ya escritas por el equipo de programadores de S.H.I.E.L.D como resultado de la práctica anterior. El proyecto NetBeans está disponible en el siguiente enlace: http://bit.ly/poouja1617pr8

Ejercicios

1. Modifica el método ***getCapacidadDestructiva, toCSV*** y el *destructor* de la clase *Poder* para que sean **virtuales**, de forma que se permita el polimorfismo de métodos en tiempo de ejecución en caso de que sean redefinidos en las clases derivadas.

2. Modifica la clase ***PoderPsiquico*** para que cada poder de este tipo tenga un atributo real, ***_lucidez***, con valores entre 0 y 1, que permita modificar proporcionalmente su capacidad destructiva. Implementa los métodos get/set para modificar este factor y redefine el método

getCapacidadDestructiva en esta clase para que devuelva la capacidad destructiva real afectada por este nuevo modificador (es decir: _capacidadDestructiva * _lucidez).

3. **Para la clase PoderPsiquico, adapta los constructores** y redefine cualquier otro método heredado cuyo comportamiento consideres que variará con las modificaciones realizadas ¿asignación?, ¿setCapacidadDestructiva?, ¿toCSV?, etc.

4. Implementa en la clase Mutante el **operador de comparación "menor que"** (<). Este operador utilizará como criterio de comparación la capacidad destructiva total de los mutantes que compare (es decir: un mutante será "menor que" otro si su capacidad destructiva es menor).

5. Crea, en un fichero *ordena.h*, una versión de la plantilla de función *ordena(T vector[], int tamv)* estudiada en teoría que, en vez de ordenar un vector de objetos de tipo T, ordene un vector de punteros a objetos del tipo T del estilo ***ordena(T* vector[], int tamv)***. Ten en cuenta que, a diferencia de la primera, deben compararse los datos apuntados por cada puntero en cada posición del vector y no el dato almacenado en dicha posición.

6. Implementa una **función recuperaMutantesCSV(Mutante* v[], int tamv, std::string nombreArchivo)** para leer hasta tamv mutantes del archivo indicado en formato CSV y almacenarlos en el vector indicado. La función devolverá también como resultado el número de mutantes recuperados finalmente. Hay que tener en cuenta que solo se leerán mutantes del archivo hasta que se alcance el tamaño máximo indicado o hasta que se acabe el archivo. Si quedaran más mutantes en el archivo, estos serían descartados.

7. **Revisa la implementación del constructor de copia y del operador de asignación** de la clase *Mutante*. A raíz de los cambios introducidos en la clase *PoderPsiquico*, ¿consideras que el comportamiento de dichos métodos es correcto ahora? ¿por qué? (veremos cómo solucionar este inconveniente a lo largo del Tema 4).

8. *(Opcional)* Realiza los cambios que sean necesarios en la implementación de la Jerarquía de Poderes para que sea posible **ordenar un vector de punteros a Poderes** de cualquier tipo utilizando la plantilla del ejercicio 5.

9. Modifica el archivo con la función **main**:

    9.1. Utiliza la función del apartado 7 para **leer los mutantes almacenados de un fichero** en formato CSV obtenido en la práctica anterior.

    9.2. Crea una **función de utilidad visualiza(Mutante* v[], int tamv)** que muestre toda la información de un vector de punteros a mutantes en formato CSV. Úsala para comprobar si los mutantes se han leído correctamente del archivo en el apartado anterior.

    9.3. Crea un **nuevo PoderPsíquico,** *Asfixia,* que permite infringir hasta 600 puntos de daño a un oponente biológico. Modifícalo para que su *lucidez* esté reducida a 0.75 y añádelo a uno de los mutantes del vector. Haz que se muestre la capacidad destructiva total de ese mutante antes y después de añadirle el nuevo poder.

    9.4. **Haz que se ordene el vector de punteros a mutantes según la capacidad destructiva total** de cada uno usando la plantilla de función *ordena* y se vuelva a

mostrar el listado de mutantes para comprobar que refleja el nuevo orden establecido.

9.5. *(Opcional)* Crea un **vector de punteros a poderes** y añádele Poderes de diferentes tipos: Poder, PoderFisico y PoderPsiquico, con capacidades destructivas diferentes. Haz que se muestre por pantalla el listado de poderes generado, se **ordene el vector por su capacidad destructiva** y se vuelva a visualizar para comprobar que la ordenación se ha realizado correctamente.

9.6. Libera los recursos que sea necesario.

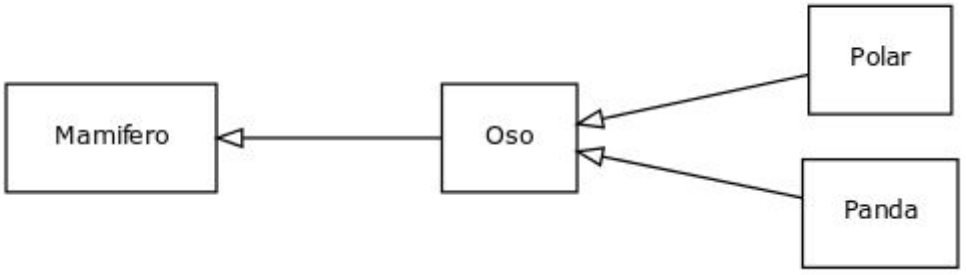
Contenidos

Objetos polimórficos

Como se ha comentado en teoría (Tema 4), el hecho de que una clase herede de otra convierte a cada objeto de dicha clase en un objeto polimórfico. Es decir, un objeto de dicha clase derivada puede comportarse, dependiendo del contexto, como si perteneciera a cualquiera de las clases de las que hereda.

Así, en la jerarquía de Mamífero mostrada en el UML de la siguiente figura, tendríamos que:

- Todo objeto de tipo "Oso" es también un "Mamífero"
- Todo objeto de tipo “Panda“ es también un objeto de tipo “Oso” y un objeto de tipo "Mamífero"



Los objetos polimórficos son muy versátiles debido a que se pueden comportar como objetos de distinto tipo al mismo tiempo. Así, podríamos definir, por ejemplo, objetos y vectores **de punteros a objetos** de tipo Mamífero, Oso y Panda:

```
Mamifero* vMamiferos[10];
Oso*  vOsos[20];

Mamifero un_mamifero;
Oso yogui;
Panda po;
```

y hacer asignaciones como las que muestran los siguientes ejemplos:

```
vMamiferos[0]=&un_mamifero;
vMamiferos[1]=&yogui // Cualquier Oso es también Mamífero
vMamiferos[2]=&po;   // Cualquier Panda es también Mamífero
```

```
vOsos[3]=&yogui;
vOsos[8]=&po; /* Cualquier objeto Panda es de tipo Oso */
```

Como sabemos, una clase derivada hereda todos los atributos y métodos de sus clases ancestros, aunque el acceso a ellos dependerá del tipo de visibilidad que estos tengan en las clases donde están definidos. Además, en ciertas situaciones, es conveniente redefinir algunos de estos métodos para adaptar su comportamiento a las características específicas de la nueva clase derivada. Sin embargo, a veces el nuevo método necesita hacer referencia expresa a la implementación original del método heredado como parte de la nueva funcionalidad que se desea implementar en la clase derivada. En este caso, para llamar al método de la clase padre, habría que utilizar su espacio de nombres para evitar una llamada de tipo recursivo al propio método que se está redefiniendo:

```
std::string Oso::toCSV () const {
  std::string cadena;
   //cadena=toCSV(); //Error. Daría lugar a una llamada recursiva
     cadena=Mamifero::toCSV() + "; "; //Correcto: representación CSV
de un mamífero
   // añadir a cadena atributos específicos de un Oso
     return cadena;
}
```

Polimorfismo de métodos

El polimorfismo de métodos es posible en C++ gracias al concepto de **enlace dinámico,** es decir, a la posibilidad de que se resuelva en tiempo de ejecución cuál es la implementación de un método específico. Esto se traduce en que, cuando tenemos un puntero o referencia a un objeto polimórfico, será el propio objeto el encargado de resolver, en tiempo de ejecución, cuál es la implementación del método que debe ejecutar.

A diferencia del polimorfismo de objetos (que ocurre cuando los objetos de clases derivadas se comportan como objetos de superclases al acceder a ellos usando variables, punteros o referencias de la superclase) **el polimorfismo de métodos es posible únicamente a través de punteros o referencias a objetos de las superclases**. Sin embargo, para que el polimorfismo de métodos se resuelva en tiempo de ejecución, es necesario que las declaraciones de los métodos estén precedidas de la palabra **virtual.** Esto permitirá que el enlazado sea realmente dinámico (en tiempo de ejecución) ya que, en otro caso, el enlace será estático. A los métodos así declarados se les denomina **métodos virtuales**. A partir de la versión 11 de C++, las redefiniciones de métodos virtuales pueden llevar opcionalmente, después de sus parámetros, la palabra reservada **override** para identificar que se tratan de métodos virtuales de una clase ancestro redefinidos en la clase actual.

Los métodos virtuales son utilizados cuando tenemos implementaciones diferentes, en clases derivadas, de un mismo método de una clase ancestro común. Estas implementaciones reflejan el comportamiento diferente que se espera del método dependiendo de la clase a la que pertenezca el objeto al que se envía dicho mensaje.

En el ejemplo de Mamífero y sus subclases, si no se utilizaran métodos virtuales y se hiciera uso de polimorfismo de objetos para referenciar objetos de las clases derivadas usando punteros o referencias de la superclase, se ejecutarían las implementaciones de los métodos de la superclase.

Sin embargo, como hemos redefinido explícitamente los métodos en la clase derivada, queremos que la implementación específica a ejecutar sea la de la clase derivada, a pesar de que el objeto de la clase derivada se referencie con un puntero o referencia de la superclase. Es decir, queremos que sea el propio objeto el que determine qué implementación debe ejecutar y no el compilador según el tipo de la referencia usada. El siguiente código muestra cómo habría que declarar e implementar los métodos:

```
/* ---
Ejemplo de polimorfismo
--- *

class Mamífero {
public:
   string metodoEstatico()  {
       return “Estático: Soy un mamífero";
   }
   virtual string metodoDinamico() {
      return "Dinámico: Soy un mamífero";
   }
};
class Oso: public Mamifero {
public:
   string metodoEstatico()  {
       return "Estático: Soy un Oso";
   }
   virtual string metodoDinamico() override {
       return "Dinámico: soy un Oso";
   }
};
int main() {
    //Usamos polimorfismo de objetos
    Mamifero *pMami=new Oso();

   // Visualiza: "Estático: soy un Mamífero"
   // Usa enlace estático
   cout << pMami->metodoEstatico() << endl;

   // Visualiza: "Dinámico: soy un Oso"
   cout << pMami->metodoDinamico() << endl;
}
```

El uso de métodos virtuales es frecuente cuando utilizamos algún tipo de contenedor que, haciendo uso de polimorfismo, utiliza punteros a objetos de diferentes clases derivadas de una superclase común. Así, en el siguiente ejemplo *figuraCompuesta* es un contenedor, y almacena 3 punteros a objetos de tipo Figura. En este caso, los objetos en cuestión deberán tener métodos virtuales con implementaciones personalizadas que permitan desde referencias a objetos de la superclase operar indistintamente con uno u otro **independientemente de su tipo real**. Si los métodos no fueran virtuales, como hemos visto, al utilizar las referencias a objetos de la superclase se ejecutarían los métodos de ésta y no los específicos de cada objeto.

```
/* ---
    Ejemplo métodos virtuales
--- */
class Figura {
```

```
int main() {
    float areatotal=0;
    Figura *figuraCompuesta[]={
          new Circulo(10),
```

```
  public:
    virtual float area() { return 0; }
};

class Circulo: public Figura {
  float radio;
 public:
  Circulo(float nradio):radio(nradio)
  {  }
  virtual float area() override{
       return PI*radio*radio; }
  };

class Cuadrado: public Figura {
   float lado;
  public:
   Cuadrado(float nlado):lado(nlado)
{}
    virtual float area() override {
         return lado*lado;
    }
};
```

```
         new Cuadrado(20),
         new Circulo(20)
   };

  for (int i=0; i<3; i++) {
areatotal+=figuraCompuesta[i]->area();
  }
  cout << "El area total de la "
      <<figura compuesta es"
      << areatotal << endl;
  /* … */
  return 0;
}
```

Observe cómo en este caso, al utilizar polimorfismo de objetos diferentes usando un vector de punteros a objetos de la superclase, sólo haciendo uso de ligadura dinámica es posible acceder a la implementación detallada de cada método dependiendo del objeto de la clase derivada que recibe el mensaje. ¿Qué se calcularía si los métodos no fueran virtuales?

Destructores virtuales

Cuando una clase implementa algún método virtual es debido a que se pretende hacer uso de polimorfismo de objetos a la hora de referenciar objetos de dicha clase. En este caso, es importante tener en cuenta que, para que se llame el destructor adecuado para dichos objetos, este debe ser implementado como virtual. Si no fuera así, si el objeto a destruir estuviera referenciado por un puntero o referencia a un objeto de la superclase, entonces sólo se llamaría el destructor de la superclase, por lo que, en determinadas circunstancias podría dar lugar a que, **al no ejecutarse el destructor de la propia clase derivada, no se liberaran los recursos que ésta pudiera haber reservado.**

A la hora de decidir si un destructor debe ser o no declarado como virtual, hay que tener en cuenta la siguiente regla: "*Si existe o existirá algún método virtual en una clase, entonces el destructor de dicha clase debe declararse de tipo virtual*". O lo que es lo mismo, si los objetos de esa clase se utilizan o van a utilizarse de forma polimórfica, entonces el destructor debe ser virtual.

En el Tema 4 se ha estudiado un ejemplo concreto donde se ponía de manifiesto esta situación.

Manejo de plantillas

Las plantillas (*templates*) de C++ son el mecanismo sintáctico que aporta el lenguaje para realizar programación genérica. En el caso particular de C++, el uso de plantillas se restringe a la implementación de plantillas de función y de plantillas de clase. Para esta práctica nos centraremos únicamente en las primeras.

Una **plantilla de función** sirve para definir la implementación de una función que puede contener uno o más tipos parametrizados (además de otros parámetros con valor constante, como se estudió en teoría). A la hora de utilizarla llamándola dentro de nuestro código, el compilador consultará los tipos de los parámetros reales que se le proporcionan en la llamada, e intentará realizar una instanciación de la correspondiente plantilla (si no se hubiera creado ya previamente), sustituyendo el tipo o tipos parametrizados de su definición por los tipos de los parámetros actuales de la llamada. La sustitución de tipos es un proceso previo a la compilación y, por tanto, meramente sintáctico. Una vez realizada la sustitución, el compilador procede a compilar el código generado automáticamente. Esto quiere decir que, aunque se haya realizado la sustitución sintáctica por los parámetros actuales, a la hora de compilar podrían producirse errores si el nuevo fragmento de código no fuera compatible por algún motivo con el nuevo tipo (o tipos) utilizado.

```
/**Ejemplo de plantilla: Intercambia dos variables de cualquier tipo
  @pre el tipo T tiene sobrecargado el operador de asignación*/
template<typename T>
void intercambia(T &a, T &b) {
    T c=a;
    a=b;
    b=c;
}
```

Este ejemplo, como se ha estudiado en teoría, es posible instanciarlo para variables de cualquier tipo de datos, lo que dará lugar en tiempo de ejecución a diferentes copias de dicha plantilla instanciadas para cada uno de los tipos necesarios:

```
int valor1=0, valor2=100;
intercambia(valor1,valor2);

Oso oso1("Yogui"), oso2("Po");
intercambia(oso1,oso2);
```

En principio podría parecer que una plantilla puede instanciarse para cualquier tipo T que se desee. No obstante, esto no es cierto, puesto que en ciertos casos es posible que el código de la plantilla necesite que se puedan realizar cierto tipo de operaciones con los datos del tipo genérico instanciado T. En clase estudiábamos el caso particular de una plantilla de ordenación de un vector por el método de selección:

```
/**Ordena un vector de elementos de menor a mayor utilizando el algoritmo de selección
  @pre Los elementos tienen sobrecargado el operador < */
template<typename T>
void ordena( T valores[], int numValores) {
    int posMenor;

    for (int i = 0; i < numValores-1; i++) {
        posMenor=i;
        for (int j = i+1; j < numValores; j++) {
            if (valores[j] < valores[posMenor])
                posMenor=j;
        }
        intercambia(valores[i],valores[posMenor]);
    }
}
```

Esta plantilla de función se instanciará adecuadamente en los siguientes casos:

```
int datos[]={1,2,3,4}
float valores[]={3.4, 5.6, 2.33, 6.55, 4.55};

ordena(datos,4);
ordena(valores,5);
```

Sin embargo, generaría un error en tiempo de compilación si intentáramos una instanciación del tipo:

```
Persona alumnos[]={Persona("Ana"),Persona("Luis")};
ordena(alumnos,2);  // ¡¡ERROR!! No está sobrecargado el operador < para Persona
```

El error de compilación es debido a que, por defecto, no existe una sobrecarga del operador < para trabajar con objetos de tipo Persona. Por lo tanto, para conseguir que nuestro código funcione correctamente, tendríamos que realizar previamente la definición específica de dicho operador (es decir <) para el tipo Persona:

```
/**
* @brief Compara dos personas por su apellido
* @param otra (Persona) Referencia a un objeto de tipo Persona
* @return true si el apellido de la persona va antes en orden alfabético que el de la
otra
*/
bool Persona::operator< (const Persona &otra) {
   return _apellidos < otra._apellidos;
}
```

Recuperación de datos de archivos de texto

En la sesión anterior estudiamos cómo utilizar la clase *std::ofstream* para almacenar información en un archivo. En particular, vimos cómo volcar una colección de objetos almacenados en un vector a un archivo, almacenando cada objeto en formato CSV en una línea diferente del archivo en cuestión.

En esta sesión, veremos cómo recuperar estos datos desde el archivo y usarlos para inicializar correctamente los objetos de nuestro programa. Para ello haremos uso de la clase *std::ifstream* que nos permitirá crear un objeto *stream* vinculado al archivo que queramos leer y utilizar sus métodos para recuperar las diferentes líneas de texto que lo componen. Para cada línea de texto, la usaremos para inicializar correctamente un objeto de tipo Plato creado usando su método *fromCSV(cadena).*

```
int main(int argc, char** argv)
{
   string cad, nombreArchivo="platos.csv";
   int cont=0;
   Plato primeros[20];   // Array de objetos inicializados por defecto

   // Instanciamos el objeto flujo de entrada
   ifstream fe;

   // Asociamos el flujo con un archivo y lo abrimos
   fe.open( nombreArchivo.c_str() );
```

```
    if ( fe.good() ) {
       // Mientras no se haya llegado al final del archivo
       while( !fe.eof() ) {
          getline( fe, cad );       // Toma una línea del archivo
          if (cad!="") {             // Ignoramos líneas en blanco
          // Pasa la línea a uno de los objetos del array, para que la
          // procese y copie los datos que contiene en sus atributos
              platos[cont++].fromCSV(cad);
              cad=""; //evitamos introducir dos veces la última línea
          };
       }

       // Cerramos el flujo de entrada
       fe.close();
    }
    return 0;
}
```

Para más detalles sobre la manipulación de flujos de texto se recomienda repasar los apuntes de C++ de la asignatura Fundamentos de Programación y el capítulo 28. *Flujos y archivos en C++*. en el libro de Joyanes (2014)