WUOLAH

POLIMORFISMO-TODO-lo-que-NECESIT...

JesusLagares

Programación Orientada a Objetos

2º Grado en Ingeniería Informática

Escuela Superior de Ingeniería
Universidad de Cádiz

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 conocimientos teóricos (con ejemplos hechos a mano) para que entiendas todos los conceptos relacionados con polimorfismo, puedas hacer los ejercicios, y aprobar el dichoso examen. No obstante, tengo que decirte que soy un alumno de informática como tú, por lo que, quizás en estos apuntes 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

Definición de Polimorfismo

El **polimorfismo** es una técnica de programación que permiten algunos lenguajes como C++ y que nos permite tratar objetos de clases relacionadas de una forma genérica, facilitando de esta manera la reutilización ahorrándonos escribir líneas de código. Gracias a esta técnica podemos lograr que un componente se comporte de una forma determinada en función del contexto.

Por ejemplo, si tenemos una *clase Instrumento* y dos *subclases: Bombo y Triángulo*, y a su vez tenemos un método *void nombre(),* que mostrase por pantalla el nombre del instrumento, podríamos implementar este método con un **comportamiento polimórfico** haciendo que, cuando se tratase de un *objeto Bombo* mostrase “Bombo”, y cuando se tratase de un *objeto Triángulo* mostrase “Triángulo”-

```
class Instrumento
{

    public :

        virtual void nombre() ; // Método polimórfico

    private :

} ;

int main ()
{

    Bombo b ;
    Triangulo t ;

    b.nombre() ;
    t.nombre() ;

    return 0 ;
```

```
class Bombo: public Instrumento
{

    public :

        void nombre()
        {

            cout << "Bombo" ;

        }

    private :

} ;
```

```
class Triangulo: public Instrumento
{

    public :

        void nombre()
        {

            cout << "Triangulo" ;

        }

    private :

} ;
```

Jesús Lagares Galán

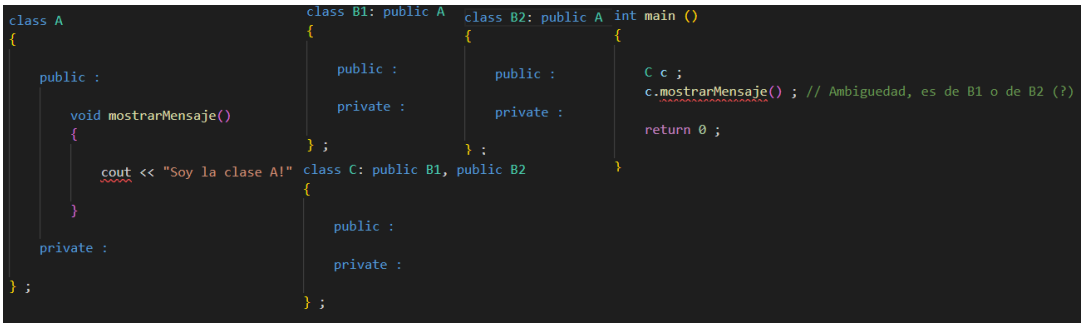
Podemos tener diferentes tipos de polimorfismo:

Polimorfismo de Sobrecarga (en tiempo de compilación)

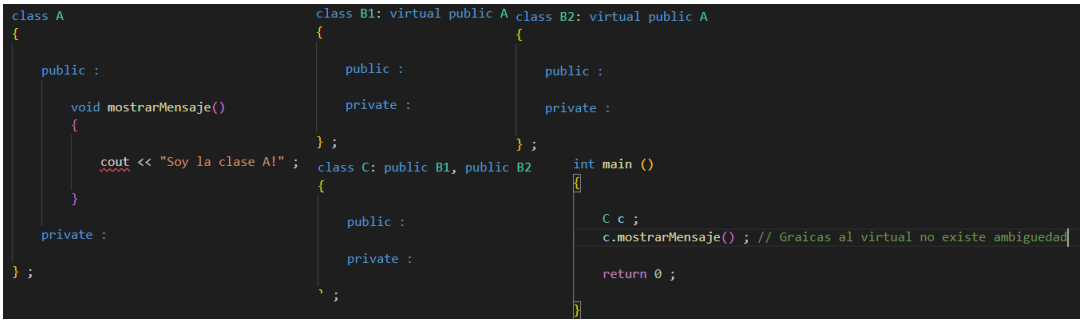
Cuando sobrecargamos un método, lo que realmente estamos haciendo es **crear un comportamiento diferente en otro contex**to, por ejemplo, cuando se le dan unos parámetros diferentes, por lo que estamos **creando un polimorfismo**. Lo que hacemos es crear un método que se comportará diferente en función de los parámetros con los que lo llamemos.

Polimorfismo En Tiempo de Ejecución

Cuando tenemos casos de **herencia múltiple**, es posible que una subclase reciba varias veces el mismo método procedente de una clase base, o también podría darse el caso en el que un método se sobrecargue a lo largo de la herencia. En ambos casos, cuando llamamos a este método desde la clase hoja (clase que no tiene ninguna subclase), se producirá una **ambigüedad**, ya que no se sabrá a qué método hay que llamar. Por este motivo, se utiliza la **palabra reservada virtual**, generando de esta manera un **polimorfismo en tiempo de ejecución**.



Si un método se repite a lo largo de una herencia múltiple, y colocamos la palabra **virtual** para **instanciar la herenc**ia, la invocación del método utilizando un objeto de su clase derivada llamará al método de la clase derivada, y en caso de que este no exista, al de la clase base.



Jesús Lagares Galán

Memorup®

ENERGY

- Rendimiento intelectual y físico
- Memoria, concentración y energía

A su vez, podemos instanciar cualquier método de la clase base como virtual, menos los constructores (LOS DESTRUCTORES SÍ PUEDEN SER VIRTUALES). Generando de esta manera una clase abstracta.

Clase Abstracta

Una **clase abstracta** es una clase de la que NO se pueden crear objetos porque **alguna de sus operaciones no se encuentra definida** (a estas operaciones las denominamos operaciones abstractas). Una operación de este tipo, es decir, una operación abstracta es aquella que no se implementa, tan solo **se define utilizando un método virtual puro** (método virtual igualado a 0).

A estos métodos también se les denomina **métodos diferidos**, porque es como si dijésemos: "Este método ya se definirá en otro sitio (en sus clases derivadas)".

Una clase que cuente con, al menos, un método abstracto, será una clase abstracta. Estos métodos abstractos deberán ser implementados en las subclases.

Por ejemplo:

```
class Instrumento // Clase abstracta
{

    public :

        virtual void nombre() = 0 ; // Método abstracto o método virtual puro

    private :

} ;
```

```
class Bombo: public Instrumento
{

    public :

        void nombre()
        {

            cout << "Bombo" ;

        }

    private :

} ;
```

Interfaces

Si una clase tiene todos sus métodos abstractos, esta se denominará **interfaz**.

Una asociación entre una interfaz y una clase se denomina **realización**.

Identificación de tipos en tiempo de ejecución

A la hora de trabajar con herencia, no debemos olvidar que estamos trabajando con **tipos y subtipos**. Por este motivo, pueden darse problemas durante la ejecución. No obstante, C++ nos provee de mecanismos para determinar con seguridad tipos en tiempo de ejecución, tanto para hacer conversiones como para determinar el tipo real al que referencia un puntero.

Estos mecanismos son:

- ### dynamic_cast

  dynamic_cast<tipo>(expresión).

  Gracias a esta instrucción podemos comprobar **si es posible hacer una conversión** de un valor (expresión) a un tipo determinado (puntero). Si la conversión es válida, se devuelve el valor requerido; sin embargo, si devuelve 0 o lanza la excepción bad_cast la conversión no es válida. Con este operador garantizamos que la conversión se realiza sí, y solo sí, si el objeto es del tipo del destino. Por ejemplo:

```
Base *pb;

//Explicacion de DYNAMIC_CAST
Derivada* pd=dynamic_cast<Derivada*>(pb);
/*Estamos convirtiendo un puntero de tipo Base a un puntero
  de tipo  Derivada, esto es posible gracias a que la clase
  Derivada heredad de la clase Base*/

//Sin embargo si tratamos de hacer lo siguiente:
Persona *pepe = dynamic_cast<Derivada*>(pb);
/*La clase Persona no deriva de Base por tanto no se podra hacer esto*/

if(pd != nullptr)
    std::cout<<"Conversion correcta";
else
    std::cout<<"Conversion fallida";

/*El dinamic_cast permite que si no se lleva a cabo la conversion
 simplemente el puntero apuntara a nullptr y el programa no temrminara
 sin embargo static_cast hara que el programa termine si no se puede
 llevar a cabo la conversion.*/
```

Utilizamos este mecanismo para **comprobar la jerarquía de clases de una herencia**. Por ejemplo, si tenemos una clase B, y una clase derivada de B llamada D, podemos hacer una función que procese un objeto de B y que trate de forma especial los que son de tipo D:

Supóngase los siguientes tipos polimórficos:

```
class B {
public:
  virtual ~B() {}
  // ...
};

class D: public B {
  //...
};
```

La siguiente función procesa a través de un puntero un objeto perteneciente a la jerarquía de clases de raíz *B*. Sin embargo, trata especialmente los que son en realidad de tipo *D*.

```
void procesar(B* pb)
{
  if (D* pd = dynamic_cast<D*>(pb)) {
    // El objeto apuntado por «pb» es de tipo «D»
    // Así «pb» se convierte sin problemas en «pd»
  }
  else {
```

Mediante este mecanismo también realizamos un **modelado "hacia abajo"**, algo que era inseguro con conversiones explícitas; sin embargo, al utilizar dynamic_cast, el programa terminará si se produce cualquier tipo de error, por lo que es más seguro realizar esta conversión "hacia abajo" que con un modelado ordinario.

- **static_cast**

static_cast<tipo>(expresión).

Este es el operador de conversión utilizado por defecto, se encarga de comprobar en tiempo de compilación que los tipos de origen y destino sean compatibles (por eso lo de *static*). Sin embargo, este operador no realiza ningún tipo de comprobación o lanza alguna excepción, por lo que **podríamos acabar teniendo punteros no válidos**.

- **const_cast**

const_cast<tipo>(expresión).

Gracias a este operador podemos **modificar una variable constante**. Creamos una "puerta trasera" para modificar esta variable gracias al uso de un puntero

```
const int n = 1;

int *p = const_cast<int*>(&n);

*p = 2;
```

- **typeid**

typeid(objeto/referencia/puntero/nombreDeTipo)

```
typeid(r);  // tipo del objeto referido por 'r'
typeid(*p); // tipo del objeto al que apunte 'p'
typeid(p);  // tipo Forma*; válido pero evidente
```

typeid() es una instrucción contenida en la librería **<typeinfo>** que nos devuelve por referencia un **objeto no modificab**le (const) **de tipo type_info**, en este tipo estará contenido el tipo del objeto al que refiere el argumento que le damos (ya sea un objeto, una referencia, un puntero…). Los objetos que devuelve esta instrucción se pueden comparar por == y ¡=. Gracias a esta instrucción podemos determinar el tipo durante un polimorfismo.

```
void tocar(Instrumento& i )
{

    if ( typeid(i) == typeid(Bombo) )
    {

        cout << "Está sonando un bombo" ;

    }

    else if ( typeid(i) == typeid(Triangulo) )
    {

        cout << "Está sonando un triángulo"

    }


}
```

¿Cuándo utilizar cada uno de ellos?

- **Static_cast** suele uitlizarse para hacer conversiones estáticas entre tipos definiddos por el lenguaje (de int a float, por ejempl). Además de esto, podemos utilizarlo en la herencia para convertir hacia arriba, pero debemos tener cuidado si queremos convertir hacia abajo, ya que static_cast no realiza ningún tipo de comprobación, por lo que puede retornar punteros indefinidos o no válidos.

- **Dynamic_cast** para **convertir hacia abajo**, ya que nos brinda una excepción y, en el peor de los casos, devolverá un nullptr. Aunque realmente se utiliza siempre que estemos manejando polimorfismo, ya que, además de hacia abajo, también nos permite convertir hacia arriba, e incluso hacia los lados si existiese en la jerarquía de clases. La diferencia a la hora de utilizar typeid() y dynamic_cast para determinar el tipo en una jerarquía de clases radica en que, utilizando dynamic_cast podemos acceder a métodos únicos de subclases; en cambio, si utilizamos typeid() el compilador no nos va a dejar. Dynamic_cast, a diferencia de typeid, nos permitirá realizar una conversión de punteros (o referencias) entre tipos pertenecientes a la misma jerarquía de clases. Esto se hace necesario cuando tenemos que invocar a métodos que son específicos de una clase derivada. Por ejemplo, en las prácticas, la fecha de expiración solo forma parte del LibroDigital, pero no del Articulo. Por lo tanto, si quiero invocar al observador que me devuelve esa fecha, no lo puedo hacer desde un puntero a Articulo, sino desde un puntero a LibroDigital. Y para ello se requiere dynamic_cast.

- **Const_cast** cuando queramos modificar constantes.

- **Typeid** cuando queramos identificar el tipo de objeto para trabajar de una forma u otra en función de si es de la clase base, de la subclase B, de la subclase C, pero no necesitemos acceder a ningún método de las subclases.

Polimorfismo Paramétrico

C++ nos brinda la capacidad de utilizar un **tipo de dato como parámetro** (clásico <typename T> que se ve en AAED) implementándose mediante **clases paramétricas**. Esto no es más que otro tipo de **polimorfismo**, ya que, **la clase cambiará su comportamiento en función del tipo de parámetro que reciba** (no es lo mismo mostrar por pantalla un entero que un char, por ejemplo).

Una **plantilla** no es más que una **definición genérica de una clase**. De esta manera hacemos que el comportamiento de la clase no dependa de los tipos concretos de los parámetros reales, sino que pueda adoptar un **comportamiento polimórfico** para trabajar con diferentes tipos de datos.

```
Pila<char> p1;                    // pila de caracteres
Pila<string> p2(200);             // pila de cadenas
Pila<complex<double> > p3(300);   // pila de complejos
```

Para definir una clase paramétrica utilizaremos el término: **template <typename T>,** si por ejemplo definimos una clase paramétrica A, quedaría **template <typename T> class A**. Y, por tanto, el nombre de la clase sería **A<T>**. No debemos olvidar incluir la definición de la parametrización (template <typename T>) a la hora de implementar cada miembro de la clase.

Jesús Lagares Galán

Por ejemplo:

```
template <typename T>
class A
{

    public :

        void mostrarT() ;

    private :

        T t ;

} ;
```

```
template <typename T>
void A<T>::mostrarT()
{

    cout << this->t ;

}
```

Y para utilizar este tipo de clases tan solo sustituimos T por el tipo en cuestión:

```
int main ()
{

    A<int> a ;

}
```

Puedes encontrar ejemplos de clases como **Vector<T> o Matriz<T>** dentro del libro **Fundamentos de C++.**

De igual manera, una clase paramétrica puede tener varios parámetros, encadenándose como lista de parámetros: **template <typename T1, typename T2>.** De la misma forma, se le pueden asignar valores por defecto:

```
template <typename T = char, size_t n = 256> class Buffer {
  T b[n];
  // ...
};

// ...

V<double> a;
V<> b;
```

Así, *a* será un búfer de, a lo sumo, 256 elementos de tipo *double*. Nótese cómo al definir *b* se omiten ambos parámetros de plantilla.

Jesús Lagares Galán

Pregunta de… ¿Es esta la mejor manera?

Realmente este no es un concepto relacionado con el **polimorfismo**; sin embargo, aparece repetidamente en numerosos exámenes y, aunque puede resolverse sabiendo toda la teoría de polimorfismo y pensando un poco nunca viene mal una explicación adicional 😉.

En numerosos exámenes se expondrá un enunciado en el que se implementa una función polimórfica utilizando **dynamic_cast o typeid**, y se nos preguntará si esta es la mejor forma de implementar dicha función. Expongo un enunciado de ejemplo:

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

- ¿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.

Por regla general, implementar con typeid no será la mejor manera, ya que genera una repetición de código bastante alta; en su lugar, la mejor manera de implementar el polimorfismo sería utilizar una clase abstracta que tenga como método polimórfico (utilizando la palabra reservada virtual) el método que se pretende crear (rotar() en este caso), ya que de este modo evitamos esa repetición de código que no hace más que aumentar la complejidad ciclomática del programa.

Siguiendo con el ejemplo, una solución óptima sería así:

Jesús Lagares Galán

```
class Figura // Clase abstracta
{

    public :

        virtual void rotar() = 0 ; // Método virtua

    private :
} ;

class Circulo: public Figura
{

    public :

        void rotar()
        {

            // No se hace nada

        }

    private :

} ;
```

```
class Triangulo: public Figura
{

    public :

        void rotar()
        {

            // Rotar triangulo

        }

    private :

} ;

class Cuadrado: public Figura
{

    public :

        void rotar()
        {

            // Rotar cuadrado

        }

    private :

} ;
```