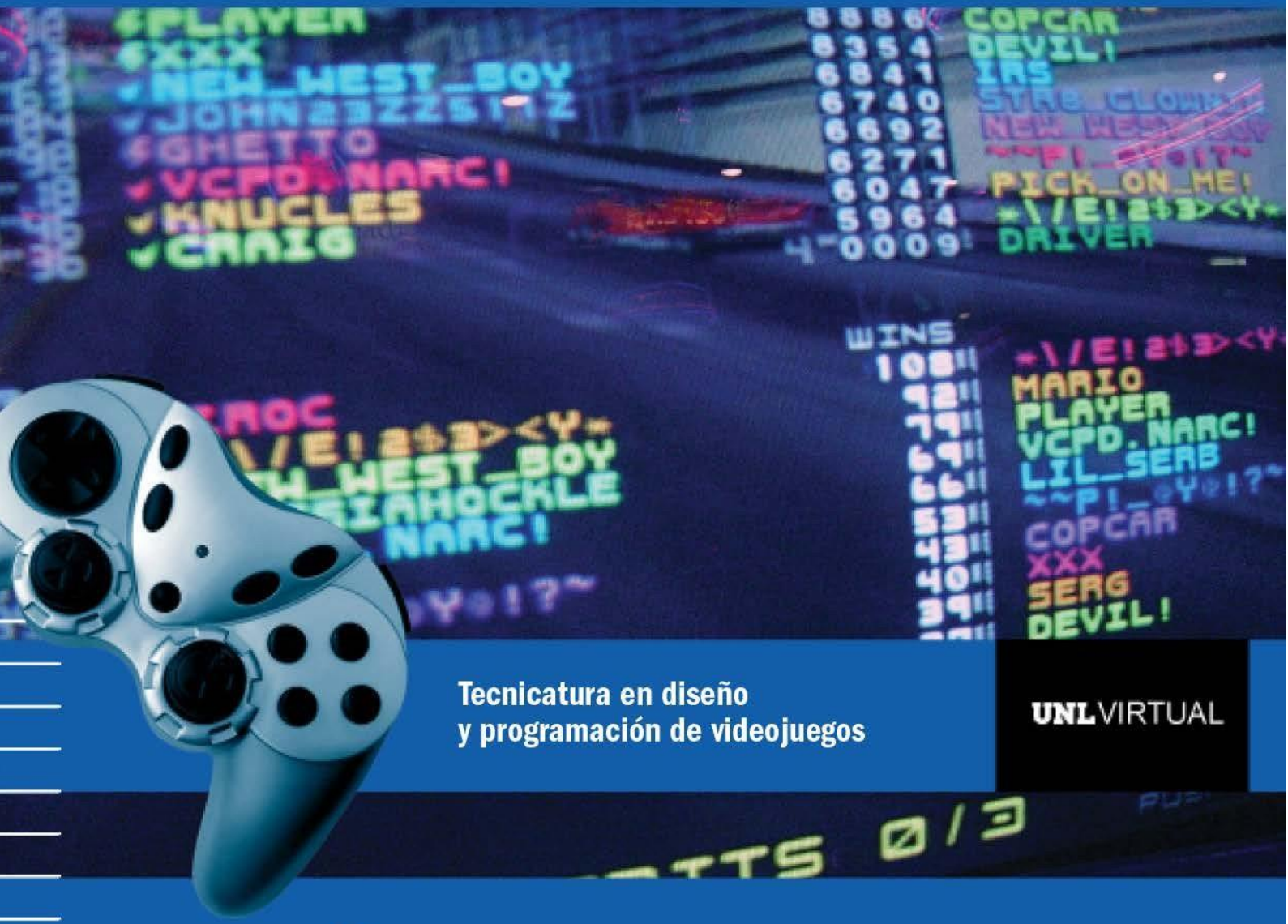
UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS



Tecnicatura en diseño y programación de videojuegos

UNL VIRTUAL

Introducción a la programación

Unidad 9
Polimorfismo

INTRODUCCIÓN

Con la herencia habíamos visto la posibilidad de ampliar una clase a partir de una o más clases madres. Incluso permitiendo configurar qué métodos y atributos compartir. Esto es un concepto esencial a la hora de utilizar objetos, ya que no sólo nos permite organizarnos adecuadamente, sino que también nos ahorra escribir varias veces el mismo comportamiento y/o código. De esta manera, podemos obtener una clase hija compuesta por los elementos que creamos necesarios. Básicamente, lo que nos permite la herencia es crear una clase armada de pedacitos de otras clases, tomando justo lo que nos hace falta.

Existen ciertas oportunidades en donde necesitamos extender, de la misma manera, una clase, pero conservando el tipo de clase, es decir, en vez de ampliar nuestra clase hija, agregarle o modificarle cosas a la madre. ¿Cuándo necesitamos esto? Esta situación es muy común cuando trabajamos con objetos ya definidos, ya sea por algún motor, una librería, alguna interfaz o nosotros mismos. Cuando utilizamos un motor de desarrollo, las clases que utilizamos para crear nuestros componentes son siempre las mismas y todas se desprenden de los mismos 3 o 4 tipos de objetos que nos ofrece el motor, pero en cada situación nosotros vamos definir el comportamiento del tipo de objeto para que se adapte a nuestras necesidades. Por ejemplo, un motor nos ofrece, entre otras cosas, trabajar con un objeto que no se mueve llamado **Estático**, que una vez fijada las coordenadas se queda fijo en la pantalla y sólo recibe colisiones de los demás. Este tipo de objeto nos sirve para crear un obstáculo, el piso, una pared, una trampa, etc. ¿Y por qué necesitamos qué conserve el tipo de dato? Bien, porque el motor ya sabe cómo se comporta el tipo de objeto, ya sabe cómo interactúa el tipo de objeto **Estático**, luego, si es una pared, una trampa, el piso, una pelota u otra cosa le es indistinto. Aunque esto no impide que nosotros, como usuarios de ese objeto, le podamos decir cómo comportarse ante particulares situaciones, si cambiar de color, desaparecer, cambiar de tamaño o el comportamiento que queramos definirle, siempre partiendo la posibilidad de redefinir su comportamiento natural.

Otro ejemplo, en cualquier juego de estrategia (Age of Empires, Warcraft, Command and conquer, etc). Por cada imperio, tribu o legión que elijamos, vamos a tener los mismos tipos de personajes con los mismos movimientos, aunque en cada caso, se va a comportar distinto. En cada caso vamos a tener un jinete, o un piquero, o un recolector.

Esto es lo que se denomina en la Programación Orientada a Objetos **Polimorfismo**. El Polimorfismo es un conjunto de técnicas que nos permiten darte a los objetos, comportamientos de distintos tipos de clases diferentes.

POLIMORFISMO EN C++

El Polimorfismo es un concepto bastante simple de aplicar, aunque su comprensión puede ser dificultosa de ver, sobre todo si recién estamos entrando en el mundo de los Objetos. Vamos a intentar verlo en varias etapas. Pensemos una situación, tenemos una clase felino que puede maullar y ronronear. Tiene un constructor pero no define nada, tampoco tiene atributos, sólo hace algunas onomatopeyas.

```
class Felino{
    public:
        Felino(){}
        void Maullar(){cout<<"miauuu"<<endl;}
        void Ronronear(){cout<<"Prrrrrr"<<endl;}
};
```

Ahora desde el main instanciamos un objeto Felino y utilizamos sus métodos.

```
int main (int argc, char *argv[]) {

    Felino feli;
    feli.Maullar();

    return 0;
}
```

La salida en consola del código anterior es:

> miauuu

Vamos a crear ahora una clase Humano, que podrá acariciar a una instancia Felino.

```
class Humano{
    public:
        Humano(){}
        void Araciar(Felino minino){
            minino.Ronronear();
        }
};
```

Desde el main:

```
int main (int argc, char *argv[]) {

    Humano Juan;
    Felino feli;
    feli.Maullar();
    Juan.Araciar(feli);

    return 0;
}
```

La salida en consola será:

> miauuu
> Prrrrrrr

A continuación sólo vamos a cambiar las definiciones estáticas por punteros:

```
class Humano{
    public:
        Humano(){}
        void Araciar(Felino *minino){
            minino->Ronronear();
        }
};

int main (int argc, char *argv[]) {

    Humano *Juan;
    Felino *feli;
    feli->Maullar();
    Juan->Araciar(feli);

    return 0;
}
```

Ahora vamos a crear una clase Tigre, Tigre hereda de Felino, y por ende podrá re implementar los métodos que consideremos necesarios. Vamos a mantener el Ronroneo cómo lo tiene definido Felino, pero modificaremos Maullar por algo más característico de un Tigre.

```
class Tigre: public Felino{
    public:
        Tigre(){}
        void Maullar(){cout<<"Grrrrrr"<<endl;}

};
```

Ahora en el main definimos feli como una instancia Felino aunque apuntaremos a un nuevo Tigre:

```
int main (int argc, char *argv[]) {

    Humano *Juan;
    Felino *feli = new Tigre();
    feli->Maullar();
    Juan->Araciar(feli);

    return 0;
}
```

Aquí vemos el Polimorfismo en esencia, en la línea:

```
Felino *feli = new Tigre();
```

Feli es una instancia de Felino y por ende tiene sus métodos y atributos, aunque apunta a un objeto Tigre definido exclusivamente para la instancia de Felino. Esto es posible porque Tigre es una clase derivada de Felino. Aquí hay dos cosas interesantes. El humano sigue actuando sobre un Felino con su método **Acariciar**, por ende, nuestro Tigre sigue siendo captable de caricias. Y también tiene la posibilidad de maullar. Ahora, aquí tenemos un problema, la salida en consola del código es:

> miauuu
> Prrrrrrr

Pero ¿Por qué? Si redefinimos el método maullar. Aquí está la segunda clave del Polimorfismo. Si nosotros queremos hacerle notar al compilador que podemos utilizar los métodos de las clases derivadas en caso de utilizar una definición como la anterior, debemos anteponer la palabra **virtual** delate del método de la clase madre:

```
class Felino{
    public:
        Felino(){}
        virtual void Maullar(){cout<<"miauuu"<<endl;}
        virtual void Ronronear(){cout<<"Prrrrrr"<<endl;}
};
```

Ahora sí, si en la clase derivada redefinimos cualquiera de los métodos virtuales (no hace falta definir como virtual todos los métodos, sólo aquellos que puedan ser redefinidos) podremos ejecutarlos desde el objeto creado como instancia de la madre. Si volvemos a ejecutar el último main, sí podremos ver el Maullar de Tigre:

> Grrrrrr
> Prrrrrrr

Vamos a crear ahora otra clase, también derivada de felino que se llama Gato, Gato redefine el método ronronear. El gato, además de ronronear echará un par de zarpazos:

```
class Gato: public Felino{
public:
    Gato(){}
    void Ronronear(){cout<<"Prrrrrr zap zap zap!!"<<endl;}

};
```

Luego en el main:

```
int main (int argc, char *argv[]) {

    Humano *Juan;
    Felino *feli = new Tigre();
    Felino *gatito = new Gato();
    feli->Maullar();
    gatito->Maullar();
    Juan->Araciar(feli);
    Juan->Araciar(gatito);
```

```
    return 0;
}
```

Gatito Maulla como Felino, ya que no redefine el método, aunque cuando el Humano acaricia al Felino, este ejecuta el método ronronear que está redefinido en Gatito, y por ende, ronronea con su propio método. La salida del código del main es:

> Grrrrrr
> Miauuuu
> Prrrrrrr
> Prrrrrrrrr zap zap zap!!

COMPORTAMIENTO DE LAS CLASES HIJAS

Bien, a este punto reforzaremos la idea que el polimorfismo tiene como objetivo principal modificar el comportamiento de una clase (la clase madre) para hacerla comportarse como una clase hija a partir del renombramiento de los métodos que la clase madre pudiera tener. Pero bien ¿Qué ocurre con los métodos y atributos propios de las clases hijas?

Le agregaremos a la clase Gato un método público (aunque podría ser privado) **dormirse**, que hace que el gato se duerma:

```
class Gato: public Felino{
public:
    Gato(){}
    void dormirse(){ cout<< "Prr zzzzzz"<<endl;}
    void Ronronear(){
        cout<<"Prrrrrr zap zap zap!!"<<endl;}
};
```

Si intentamos llamar al método dormirse desde el main:

```
int main (int argc, char *argv[]) {

    Felino *gatito = new Gato();
    gatito->dormirse();
    return 0;
}
```

El compilador dirá que la clase “Felino” no tiene ningún método “dormirse”, lo cual es cierto. No hay que olvidar que gatito es una instancia de Felino, por ende no podremos ejecutar ningún método de la clase hija. Aunque esto no nos imposibilita tener métodos y atributos propios (privados o públicos) de la clase hija que nos permitan controlar su comportamiento. ¿Pero qué hacemos con esos métodos y atributos? Todavía los podremos utilizar desde el constructor o los otros métodos. Por ejemplo, si queremos que el gato se duerma luego de ronronear, podemos hacer que éste se ejecute en el mismo método sobrecargado de la clase madre:

```
class Gato: public Felino{
public:
    Gato(){}
    void dormirse(){ cout<< "Prr zzzzzz"<<endl;}
    void Ronronear(){
        cout<<"Prrrrrr zap zap zap!!"<<endl;
        dormirse();  }
};
```

Ahora en el main:

```
int main (int argc, char *argv[]) {

    Felino *gatito = new Gato();
    gatito-> Ronronear ();
    return 0;
}
```

Y en la consola veremos:

> Prrrrrrrrr zap zap zap!!
> Prrr zzzzzzzzzzz

Quizás esto pueda parecer poco útil, pero no hay que olvidar que la clase hija sólo tiene como objetivo darle un nuevo comportamiento a la madre. Si tenemos muchos métodos nuevos propios quizás no sea correcta la aplicación de Polimorfismo.

RESUMEN FINAL

En resumen, el Polimorfismo nos permite hacer actuar un objeto como otra instancia distinta, o bien (visto de otra forma), re implementar una clase con otro comportamiento. Si se quiere, se puede decir que la herencia directa expande una clase hija, agregando comportamiento de una clase madre. En cambio, en el polimorfismo (que también es una herencia), lo que se expande es la misma clase madre, modificando el comportamiento a partir de las clases hijas que se le inyectan diferencias.

¿Por qué polimorfismo? El Polimorfismo no es mejor ni peor que la herencia directa, tiene otras características.

Permite conservar la clase base, esto es bueno ya que podemos tener montada toda una estructura que trabaja con una clase en particular (en el ejemplo Humano con Felino). Además, podemos agregar y quitar clases derivadas sin afectar la estructura principal del programa que trabaja sobre la clase madre.

En el Polimorfismo no podemos hacer que la clase madre adquiera nuevos comportamientos, si no tiene definido algún atributo o método, no podremos agregarlo desde la clase hija (sólo re implementa métodos ya declarados).

En la herencia directa podemos modificar y agregar elementos de distintas clases madres. Aunque vamos a tener una nueva clase y por ende, tendremos que tener definida las relaciones de esta clase hija con el resto del entorno.

Aquí utilizamos sí o sí punteros, aunque no es una cuestión única de Polimorfismo, se puede utilizar con Objetos y Herencia por igual.