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

CONTENIDOS

1. Polimorfismo: las mil y una formas

Cuando manejamos un auto, lo hacemos de una forma muy distinta a la que empelamos al montar una bicicleta. Sin embargo, si la vida fuese un extenso código de programación, posiblemente el auto y la bicicleta descenderían de la clase abstracta *vehículo* y heredarían el método *conducir…*

Esta capacidad de tener objetos que se comportan de manera diversa, pero que son derivados de la misma clase base y con los mismos métodos, se denomina *polimorfismo*.

Cuando vimos objetos en la unidad 2, aprendimos que al hacer uso de la palabra reservada *class,* colocábamos en un mismo nivel a la filosofía y a la sintaxis de la Programación Orientada a Objetos (POO). Es decir, si un amigo, que recién está dando sus primeros pasos en la programación nos dice: "hoy programé una pequeña clase", a pesar de que no lo sepa o lo haga mal, está realizando una programación que orienta a objetos.

Nosotros, en cambio, que en ese momento estamos programando en C (no en C++) y no tenemos la posibilidad de hacer uso de la palabra reservada *class,* tenemos que conformarnos con usar estructuras y funciones para conseguir una orientación a objetos. Esto, claro está, no llegará a tener el encapsulamiento deseado, pero la filosofía está presente.

Algo similar ocurrió también cuando vimos herencia. Si bien C++ no tiene una palabra reservada (como sí la tiene Java con *extend*), existe una sintaxis que nos define una herencia, y si hacemos uso de ella, indefectiblemente heredaremos.

Ahora bien, en polimorfismo sólo tenemos la filosofía; no existe una sintaxis ni una palabra reservada que nos indique que estamos haciendo polimorfismo. Para conseguirlo, podemos hacer uso de muchas herramientas, algunas más formales, otras más robustas, pero cualquiera nos servirá si nos ayuda a conseguir el objetivo.

En esta unidad veremos tres herramientas (quizás las más potentes) con las que contamos en el lenguaje para llegar al polimorfismo.

Sobrecarga de métodos

Posiblemente el lector esté familiarizado con este concepto, ya que lo vimos en unidades anteriores. Además, es idéntico al que usamos cuando manejamos funciones. Por esta razón no lo trataremos con demasiada profundidad, pero sí lo describiremos brevemente, dado que es una de las tres herramientas fundamentales con las que cuenta el lenguaje para conseguir el polimorfismo.

Tal como sucede cuando manejamos funciones, los métodos permiten sobrecargarse tanto dentro de la misma clase, como en las clases derivadas, aunque el efecto es diferente.

Supongamos que tenemos que programar el famoso juego de Nintendo, *Pokemon.*[1] A primera vista podríamos decir que todos los pokemones se heredan de una gran clase madre, *Pokemon*, de la cual salen sus clases hijas, basadas en los tipos de pokemon: *agua, fuego, hierba, roca,* etc., de los cuales tomaremos sólo los dos primeros.

[1] *Pokemon* es una saga de videojuegos RPG. Su primera versión *(Pokemon azul)* salió al mercado en 1996 para la consola Game Boy, también de Nintendo. Los pokemones son una suerte de animales de distintos tipos de elementos, con poderes diversos y que se utilizan para infinidad de cosas, desde combate hasta ayudantes en el hogar.

Supongamos que la clase madre *Pokemon* solamente contiene las características comunes a todos los pokemones: energía, nivel y ataques básicos comunes a todos los tipos (envestida, arañazo, rasguño). Las clases particulares a los tipos, como *agua* y *fuego,* tienen los ataques particulares de los pokemones de estos tipos.

Supongamos que queremos programar la clase *agua;* la misma tendrá los métodos *lanza_agua* y *salpicar.* Asimismo, contamos con la clase particular del *Pokemon,* que poseerá las características propias del mismo. Supongamos que el *Pokemon Larpas* es del tipo agua y que, además, tiene una modalidad de ataque llamada *rayo_larpas*y obviamente ejecuta todos los golpes comunes a los pokemones.

Un detalle que omitimos es que, además de esta clasificación, los pokemones están tipificados en base a las características físicas de los mismos: *voladores*, *excavadores*, *nadadores*, etc. Supongamos, entonces, que *Larpas* es volador, por lo cual tiene las características de los pokemones voladores, esto es, hereda el método *volar*.

La complejidad aquí reside en la clase *Larpas,* que ejecuta su rayo_larpas de dos maneras distintas. Una de las formas recibe como parámetro el nivel del oponente y devuelve el daño efectuado con el cálculo: | Nivel_Pokemon - Nivel_Op | * 2. Si la resta da negativo, devuelve sólo 2. En tanto, la segunda forma recibe como parámetro la energía del oponente y devuelve la mitad de ese valor.

El siguiente esquema define a *Larpas:*

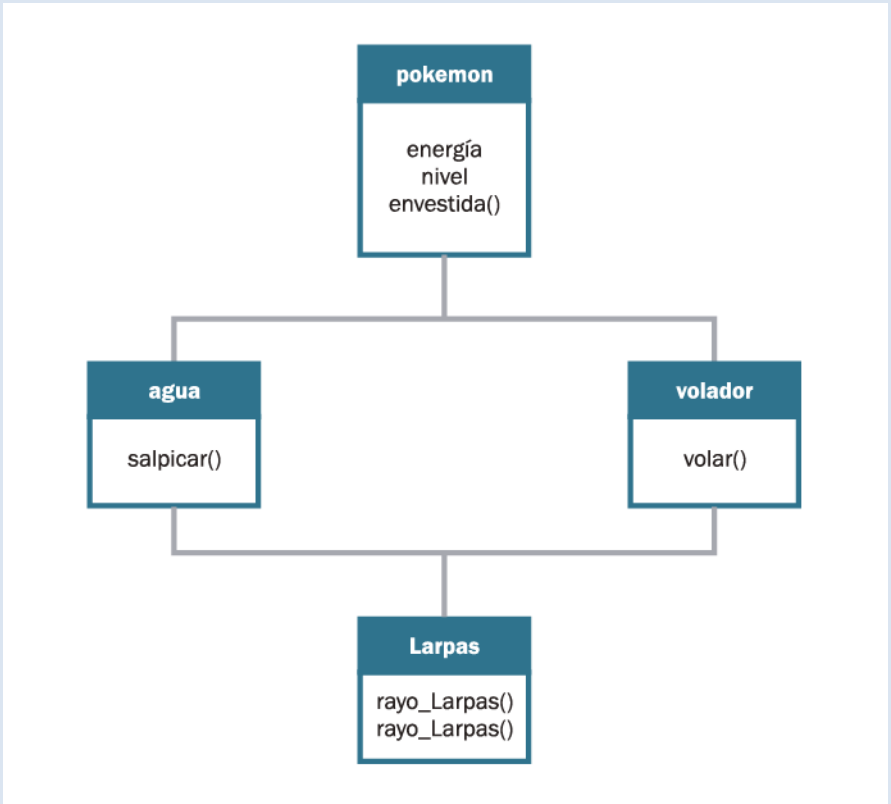

Figura 1. *Esquema que define a* Larpas.

Veamos el código de *Larpas:*

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


class pokemon {

  private:
    int nivel;
    float energia;

  public:
//      float envestida(int);
//      pokemon();

};

class agua : public pokemon {

  public:
    void salpicar();
};

class volador : public pokemon {

  public:
    void volar();
};

class Larpas : public agua, public volador {

  private:
    int nivel;
  public:
    Larpas();
    float rayo_larpas(int);
    float rayo_larpas(float);
};


void agua::salpicar(){
  cout<<"SQUASH SQUASH"<<endl;
}

void volador::volar(){
  cout<<"ESTOY VOLANDO!"<<endl;
}


Larpas::Larpas(){nivel=10;}

float Larpas::rayo_larpas(int L){

  return (nivel - L)> 0 ? (nivel - L)*2.0 : 2.0;
}

float Larpas::rayo_larpas(float E){

  return (E*0.5);
}


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

Larpas PokeLarpas;

//  rayo_larpas por nivel

cout<< PokeLarpas.rayo_larpas(5)<< endl;

//  rayo_larpas por energia
```

```
64
65  cout<< PokeLarpas.rayo_larpas((float)50.0)<< endl;
66
67  // volar y salpicar
68
69  PokeLarpas.volar();
70  PokeLarpas.salpicar();
71
72
73  return 0;
74  }
```

Script 1

Llegados a este punto, es necesario realizar algunos comentarios:

Empezando por la línea 29, esa doble herencia se denomina *herencia múltiple* y le permite a una clase derivada heredar de más de una clase base. No estudiamos esto en la unidad de herencia porque debemos conocer algunos detalles antes de hacer una herencia múltiple.

En la línea 12 y 13 tenemos el constructor de la clase *Pokemon* comentado, al igual que el método *envestida.* También, en las líneas 8 y 9, están los atributos de la clase como *private* (en calidad de *privados*), lo cual nos obligó a redefinir el atributo *nivel* en la clase *Larpas.* Esta corrección también se debe a la herencia múltiple, como acabamos de comentar. Más adelante veremos cuál es el problema que acarrea hacer una herencia múltiple y cómo solucionarlo.

En las líneas 34 y 35 declaramos los dos métodos rayo_larpas. Uno de ellos recibe como parámetro un entero y el otro, un flotante. Esta diferencia es la que le permite al compilador seleccionar el método correcto en tiempo de ejecución.

Como pueden observar, hasta aquí no hay ninguna novedad sobre lo que ya conocen acerca de funciones. Un método se puede sobrecargar todas las veces que se desee, siempre y cuando se logre diferenciarlo de los otros, variando la cantidad y el tipo de parámetros.

En C++ se puede sobrecargar tanto los métodos, como los constructores, según vimos en la unidad 2.

En la línea 65, a la hora de llamar al método rayo_larpas que recibe un flotante, se agregó entre paréntesis la palabra reservada *float,* precediendo al parámetro. Esto se llama *casteo* (una mala traducción de *casting*): se obliga al compilador a tomar ese dato como el tipo de dato que nosotros deseamos, ya que a veces suele confundirlo. Esto ocurre principalmente entre enteros y flotantes.

Herencias virtuales

Volvamos al ejemplo anterior y tomemos como caso de estudio la clase base *pokemon*.

La misma tiene el atributo *energia,* el cual es común a todos los pokemones. Lo ideal sería que este atributo fuese heredado desde esta clase y llegue hasta las clases particulares de los pokemones, como los Larpas, pasando por las clases de tipo *agua*, *fuego, roca, vegetal,* etc. y las físicas: *volador, excavador, nadador,* etc. Pero cuando le damos el permiso al atributo para que se herede de manera múltiple, nos encontramos con un problema. Veamos esto más detalladamente.

Nuevamente, nuestro diagrama de clases es el siguiente:

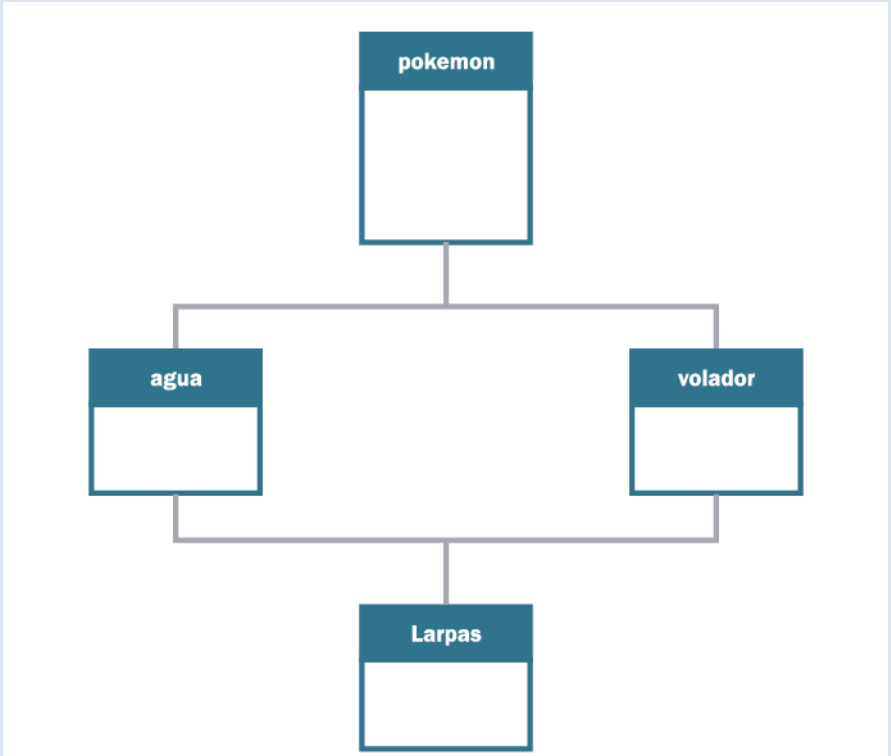

Figura 2. *Diagrama de clases.*

Si hacemos el seguimiento del atributo *energia,* podremos encontrar el gran problema que implica hacer una herencia múltiple.

Veamos:

La clase *pokemon* tiene un atributo *energia*.

La clase *agua* hereda el atributo *energia* de la clase *pokemon*.

La clase *volador* también hereda el atributo *energia* de la clase *pokemon*.

Finalmente, la clase *Larpas* hereda el atributo *energia* de la clase *agua* y también de la clase *volador*.

Es aquí, justamente, cuando surge el gran conflicto. Si bien se trata del mismo atributo que fue heredado de la misma clase base, el compilador no sabe discernir e interpreta que la clase *Larpas* tiene dos atributos *energia* idénticos. Lo mismo ocurre con los métodos: si compartimos un método que se arrastra desde la clase base *pokemon*, cuando Larpas lo herede tendrá dos métodos exactamente iguales (lo que para el compilador es una ambigüedad).

Una solución podría consistir en utilizar los atributos y métodos como privados y crear uno distinto en cada clase, pero esto es poco práctico, deficiente, denostable, entre otros, adjetivos negativos; sólo implementable en el ejemplo anterior, con la salvedad de que aún no sabíamos qué era la herencia virtual.

La mejor solución para este problema es mucho más simple, aunque no parezca, pero merece antes una buena aclaración.

Para explicarla, modifiquemos momentáneamente el ejemplo, con el fin de hacerlo más preciso a nuestro problema y más simple a la vista.

La clase *pokemon*

Atributos:

- Energía
- nivel

Métodos:

- void envestida( )
- La clase volador

Métodos:

- void volar( )
- La clase *agua*

Métodos:

- void salpicar( )
- La clase *Larpas*

Métodos:

- void rayo_larpas(int)
- void rayo_larpas(float)

Como dijimos anteriormente, si lo hacemos de la manera que sabemos, obtendremos una ambigüedad al haber una herencia dual en la clase *Larpas.* Por lo tanto, la solución consiste en anteponer la palabra *virtual* en la herencia de la clase madre.

¿Por qué sólo a la clase madre? Porque es la que está ocasionando la herencia múltiple. Si tuviésemos más de una clase con estas características, tendríamos más de una herencia virtual.

Nuestro código queda de la siguiente manera:

```
1   #include <iostream>
2   using namespace std;
3
4   class pokemon {
5
6     protected:
7       int nivel;
8       float energia;
9
10    public:
11      void envestida();
12      pokemon();
13  };

14  class agua : virtual public pokemon {
15
16    public:
17      void salpicar();
18  };

19  class volador : virtual public pokemon{
20
21    public:
```

```
22      void volar();
23  };

24  class Larpas : public agua, public volador {
25    public:
26      float rayo_larpas(int);
27      float rayo_larpas(float);
28  };


29  pokemon::pokemon(){
30    nivel=10;
31    energia=25.0;
32  }
33
34  void pokemon::envestida(){
35    cout<< "envestida pokemon"<< endl;
36  }


37  void agua::salpicar(){
38    cout<<"SQUASH SQUASH"<<endl;
39  }


40  void volador::volar(){
41    cout<<"ESTOY VOLANDO!"<<endl;
42  }


43  float Larpas::rayo_larpas(int L){
44    return (nivel - L)> 0 ? (nivel - L)*2.0 : 2.0;
45  }
46
47  float Larpas::rayo_larpas(float E){
48    return (E*0.5);
49  }


50  int main(int argc, char *argv[]) {
51
52    Larpas PokeLarpas;
53
54    //  rayo_larpas por nivel
55    cout<< PokeLarpas.rayo_larpas(5)<< endl;
56    //  rayo_larpas por energia
57    cout<< PokeLarpas.rayo_larpas((float)50.0)<< endl;
58    // salpicando Larpas
59    PokeLarpas.salpicar();
60    // envestida comun
61    PokeLarpas.envestida();
62    // volando
63    PokeLarpas.volar();
64
65  return 0;
66  }
```

Script 2

En las líneas 14 y 19 heredamos a la clase base *pokemon* como virtual, lo cual nos permitió utilizar sus métodos y atributos en la clase *Larpas* sin tener ambigüedad.

Ahora que somos programadores experimentados de herencia, veamos un tema un poco más complejo.

Métodos virtuales

Los métodos virtuales constituyen el corazón del polimorfismo, a tal punto que muchos autores los consideran sinónimos. Nosotros los veremos como una herramienta más, aunque posiblemente sea la más importante.

Como los métodos virtuales cobran importancia sólo con punteros, primero repasaremos algunas implementaciones de punteros con clases. Partamos de la declaración e implementación de la clase base *pokemon*:

```
1   class pokemon {
2     public:
3       void envestida();
4   };


5   void pokemon::envestida(){
6     cout<< "envestida pokemon"<< endl;
7   }
```

Script 3

Hasta aquí no tenemos ninguna novedad.

Veamos, ahora, cómo instanciamos la clase con punteros:

```
1   pokemon *NuevoPokemon = new pokemon();
2
3   NuevoPokemon -> envestida();
```

Script 4

En la primera línea instanciamos *NuevoPokemon,* que es un puntero del tipo *pokemon.*

Hasta aquí, entonces, no hay ningún cambio con respecto a lo que aprendimos en la unidad 1. Incluso, si no supiésemos qué es una clase, podríamos pensar que estamos frente a la instanciación de una estructura. Este nuevo puntero se iguala a una reservación de memoria y no a un dato concreto, debido a que no tenemos ningún *pokemon* creado con anterioridad para desreferenciar; si lo tuviésemos, podría ir del lado derecho de la instanciación.

En la línea 3 tenemos la llamada al método *envestida.* Aquí, quizás, reside la diferencia más notoria a la hora de utilizar punteros. El operador (**.**), que antes nos servía para acceder al método de una clase, se cambia por la flechita (->), construida por un signo menos (-) y un signo mayor (>).

Ahora sí, luego de esta breve introducción, ya estamos en condiciones de adentrarnos en los métodos virtuales.

Volvamos, entonces, al esquema de clases que obtuvimos en el *Script 2.*

Para mejorar la comprensión de los métodos virtuales agregaremos un par de clases más que permitan completar el ejemplo.

Supongamos que tenemos el pokemon Larpas, pero ese pokemos a su vez puede evolucionar varias veces, su primera evolución es Larpasone y luego puede evolucionar en Larpastwo. La cuestión importante aquí radica en que tanto Larpasone como Larpastwo siguen siendo pokemones Larpas, sólo que evolucionaron en pokemones más complejos. Normalmente hubiéramos creado a Larpasone y Larpastwo como objetos independientes heredando todo de su clase madre. El problema es que no tiene sentido redefinir un objeto que a su vez puede

cambiar, ya que los pokemones evolucionan. Esto es muy típico en los juegos, pensemos por ejemplo un juego de rol en el cual necesitamos cambiar de personaje, no es que se creamos un personaje nuevo cada vez, sólo apuntamos al que está en uso.

¿Cómo se expresa esto en la implementación? Basicamente como lo mencionamos antes, creamos una nueva clase Larpas y la referenciamos a una clase Larpasone o Larpastwo. La herencia es la que nos permite hacer eso.

A partir de este momento, cuando creemos un puntero a una clase, aprovecharemos la idea de que estamos usando una variable que puede cambiar su contenido en cualquier momento. Pensemos. Si tenemos un puntero, que puede cambiar en cualquier momento, posiblemente necesitemos que apunte a varios pokemones en todo el juego ¿Cómo conviene hacerlo? En principio, la mejor solución consistiría en crear un puntero a la clase base, asignándole luego referencias a distintos pokemones. La herencia es la que nos permite hacer esto.

```
    Larpas *NuevoLarpasOne = new Larpasone();
```

Script 5

¿Qué es exactamente esto?

La variable puntero *NuevoLarpasOne* apunta a la clase base, pero referencia a *Larpasone.*

Agreguemos, ahora, un método *envestida,* idéntico al que tiene la clase base en la clase *Larpas*, y observemos qué ocurre cuando intentamos accederlo:

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

class pokemon {

public:
    pokemon(void);
    float envestida(int);
protected:
    int nivel;
    float energia;
};

class agua : virtual public pokemon {

public:
    int salpicar();
};

class volador : virtual public pokemon{
public:
    void volar();
};

class Larpas : public agua, public volador {
public:
    virtual float envestida(int);
};

class Larpasone : public Larpas {
public:
    float envestida(int);
    float Rayo_Larpas1();
};


class Larpastwo : public Larpas {
public:
```

```
    float envestida(int);
    float Rayo_Larpas2();
};

pokemon::pokemon(){
    nivel=10;
    energia=25.0;
}

float pokemon::envestida(int L){
    cout<< "envestida pokemon"<< endl;
    return 1;
}

int agua::salpicar(){
    cout<<"SQUASH SQUASH"<<endl;
    return 1;
}

void volador::volar(){
    cout<<"ESTOY VOLANDO!"<<endl;
}

float Larpas::envestida(int L){
    cout<< "envestida Larpas"<< endl;
    return 1;
}

float Larpasone::envestida(int L){
    cout<< "envestida Larpas One!"<< endl;
    return 1;
}

float Larpasone::Rayo_Larpas1(){
    cout<< "Rayo Larpas One!"<< endl;
    return 1;
}

float Larpastwo::envestida(int L){
    cout<< "envestida Larpas Two!"<< endl;
    return 1;
}

float Larpastwo::Rayo_Larpas2(){
    cout<< "Rayo Larpas Two!"<< endl;
    return 1;
}

int main(int argc, char *argv[]) {
    Larpas *NuevoLarpasOne = new Larpasone();

    NuevoLarpasOne->envestida(5);

    return 0;
}
```

Script 5

Cuando corremos el código, notamos -con cierta decepción- que el método *envestida* que se ejecuta es el de la clase madre Larpas, cuando lo que queremos es que se ejecute el de la clase derivada.

¿Cómo solucionamos esto?

La solución está en el título del parágrafo. Si definimos como virtual al método de la clase madre, el compilador interpretará que debe ejecutar los métodos que disponga en las clases derivadas. La forma de definir al método como virtual es anteponiendo la palabra reservada *virtual* al método de la clase madre, quedando de esta forma:

```
class Larpas : public agua, public volador {
public:
    virtual float envestida(int);
};
```

Script 6

Aquí se ve la esencia del polimorfismo y esto es todo lo que hay que saber sobre métodos virtuales. Sin embargo, nos queda una larga lista de aclaraciones por hacer.

¿Qué ocurre si intentamos ejecutar desde ese mismo objeto el método *rayo_larpas1* que tenemos sólo en la clase Larpasone? No vamos a poder. El compilador nos dirá que la clase *Larpas* no tiene un método llamado *rayo_larpas1* y, por ende, desconocerá que existe en otra clase.

Esto se debe a que el puntero que creamos es un objeto que referencia a la clase *Larpas* y que tiene algunos derechos sobre la clase *Larpasone y LarpasTwo.*

¿Qué derechos? Los de acceder a aquellos métodos que están declarados en la clase madre y que son redeclarados en las clases derivadas. Luego, si logramos que dichos métodos (los de la clase madre) sean virtuales, podremos ver los de las clases derivadas.

Pero, entonces, ¿hay que declarar y definir cada método que queramos ver desde las clases derivadas? Sí y no. Sí, porque hace falta declararlos. Sin embargo, C++ nos permite evitar que los definamos, declarando dichos métodos como puramente virtuales. Por ejemplo, si quisiésemos ejecutar el método *rayo_larpas1,* tendríamos que declarar al menos la cabecera de manera idéntica en la clase madre y, en la misma declaración, igualar el método a *0 o* dejar las llaves vacías de la siguiente manera:

```
1  class Larpas : public agua, public volador {
2   public:
3    virtual float envestida(int);
4    virtual float Rayo_Larpas1(void)=0;
5    virtual float Rayo_Larpas2(void){};
6  };
```

Script 7

En la línea 4 nos encontramos con un método *rayo_larpas1* idéntico al que habíamos declarado en la case *Larpasone.* Al igualarlo a 0, C++ interpreta que ese método es puramente virtual y no necesita definirse. Lo último que queda por hacer es definir como virtuales todos los métodos que vamos a utilizar en la clase *Larpasone y Larpastwo*.

El siguiente script es un ejemplo completo de herencia múltiple, sobrecarga de métodos y métodos virtuales:

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

class pokemon {

public:
    pokemon(void);
    float envestida(int);
```

```
protected:
    int nivel;
    float energia;
};

class agua : virtual public pokemon {

public:
    int salpicar();
};

class volador : virtual public pokemon{
public:
    void volar();
};

class Larpas : public agua, public volador {
public:
    virtual float envestida(int);
    virtual float Rayo_Larpas1(void){};
    virtual float Rayo_Larpas2(void){};
};

class Larpasone : public Larpas {
public:
    float envestida(int);
    float Rayo_Larpas1(void);
};


class Larpastwo : public Larpasone {
public:
    float envestida(int);
    float Rayo_Larpas2(void);
};

pokemon::pokemon(){
    nivel=10;
    energia=25.0;
}

float pokemon::envestida(int L){
    cout<< "envestida pokemon"<< endl;
    return 1;
}

int agua::salpicar(){
    cout<<"SQUASH SQUASH"<<endl;
    return 1;
}

void volador::volar(){
    cout<<"ESTOY VOLANDO!"<<endl;
}

float Larpas::envestida(int L){
    cout<< "envestida Larpas"<< endl;
    return 1;
}

float Larpasone::envestida(int L){
    cout<< "envestida Larpas One!"<< endl;
    return 1;
}

float Larpasone::Rayo_Larpas1(){
    cout<< "Rayo Larpas One!"<< endl;
    return 1;
}
```

```
float Larpastwo::envestida(int L){
    cout<< "envestida Larpas Two!"<< endl;
    return 1;
}

float Larpastwo::Rayo_Larpas2(){
    cout<< "Rayo Larpas Two!"<< endl;
    return 1;
}

int main(int argc, char *argv[]) {
    Larpas *NuevoLarpasOne = new Larpasone();


    Larpas *NuevoLarpasTwo = new Larpastwo();


    NuevoLarpasOne->envestida(5);
    NuevoLarpasTwo->envestida(5);
    NuevoLarpasOne->Rayo_Larpas1();
    NuevoLarpasTwo->Rayo_Larpas2();

    return 0;
}
```

Script 8

Por último veremos el concepto más interesante. El hecho de utilizar punteros y referenciar una clase derivada nos da la posibilidad de modificar la referencia del objeto en cualquier momento.

¿Cómo esto?

Si tenemos un objeto Larpasone que es en definitiva un objeto Larpas y nuestro pokemon evoluciona a un Larpastwo bastará nomás con modificar la referencia del objeto.

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


    Larpas *GranLarpas = new Larpasone();

    // Creamos un Larpasone llamado GranLarpas
    GranLarpas->envestida(3);
    GranLarpas->Rayo_Larpas1();

    //GranLarpas esta evolucionando!
    GranLarpas = new Larpastwo();
    //Ahora nuestro GranLarpas es un Larpas2 y utiliza los poderes del
mismo
    GranLarpas->Rayo_Larpas2();


    return 0;
}
```

Script 9

Esta es la potencia del polimorfismo, la posibilidad de tener un objeto que cambia su estructura en cualquier momento. Esto es posible gracias a que ambos objetos fueron creados en base a una misma clase base.

Antes de escribir un código con objetos, herencias y polimorfismo debemos estructurarlo meticulosamente. La potencia de estas herramientas quedó demostrada en estas notas, pero también exigen cierta prolijidad para no terminar por embarullar más de lo que ordenamos.

El polimorfismo permite tener un esqueleto sólido en sus clases bases, con los métodos y atributos comunes a todas sus derivadas. De esta forma, agregar clases hijas es muy sencillo y completamente dinámico, ya que al quedar todo definido en las clases superiores, es posible construir objetos agregando y quitando herencias. Por ejemplo, si se nos ocurriera lograr que un pokemon de agua y volador pase de pronto a ser de fuego y terrestre.

Como conclusión podemos afirmar que la estructura de polimorfismo es una opción muy poderosa para nuestro programa y que si optamos por utilizarla, al igual que cualquier filosofía, debemos crear todo en base a ella.

2. Ejercicio integrador

Como ejercicio integrador se propone el juego *Hombres lobo contra vampiros*. Si bien es muy sencillo, lo complejizamos un poco para adaptarlo a los temas que hemos visto y poder demostrar más claramente cómo es el uso de la herencia y del polimorfismo.

Las reglas del juego son las siguientes:

- El juego es para dos jugadores.
- Los personajes son Wolverine, por parte de los hombres lobo, y Drácula, por parte de los vampiros.
- La metodología del juego es la de ataque por turnos.
- Cada jugador empieza con 40 puntos de vida y 20 puntos de energía.
- Los golpes, además de quitar vida al oponente, requieren puntos de energía.
- Cuando un golpe no tiene la suficiente energía para efectuarse, se tira un dado. Si el valor del dado *por* la energía es mayor a cierta cantidad, se efectúa el golpe; caso contrario, se pierde el turno.
- Los golpes del hombre lobo son:
    - *golpear:* quita 2 puntos de vida al oponente y toma puntos 3 de energía.
    - *morder:* quita 5 puntos de vida al oponente y toma 6 puntos de energía.
    - *aullar:* recupera 2 puntos de energía.
    - *hablar:* recupera 3 puntos de energía.
- Los golpes del vampiro son:
    - *golpear:* quita 2 puntos de vida al oponente y toma puntos 3 de energía.
    - *morder:* quita 5 puntos de vida al oponente y toma 6 puntos de energía.
    - *volar:* recupera 2 puntos de energía.
    - *hablar:* recupera 3 puntos de energía.
- Cuando se tienen 10 puntos o menos de vida, el jugador puede transformarse en su versión *Super.* Esta modalidad le permite causar doble daño en la mordida.

Diseño de clases del ejercicio:

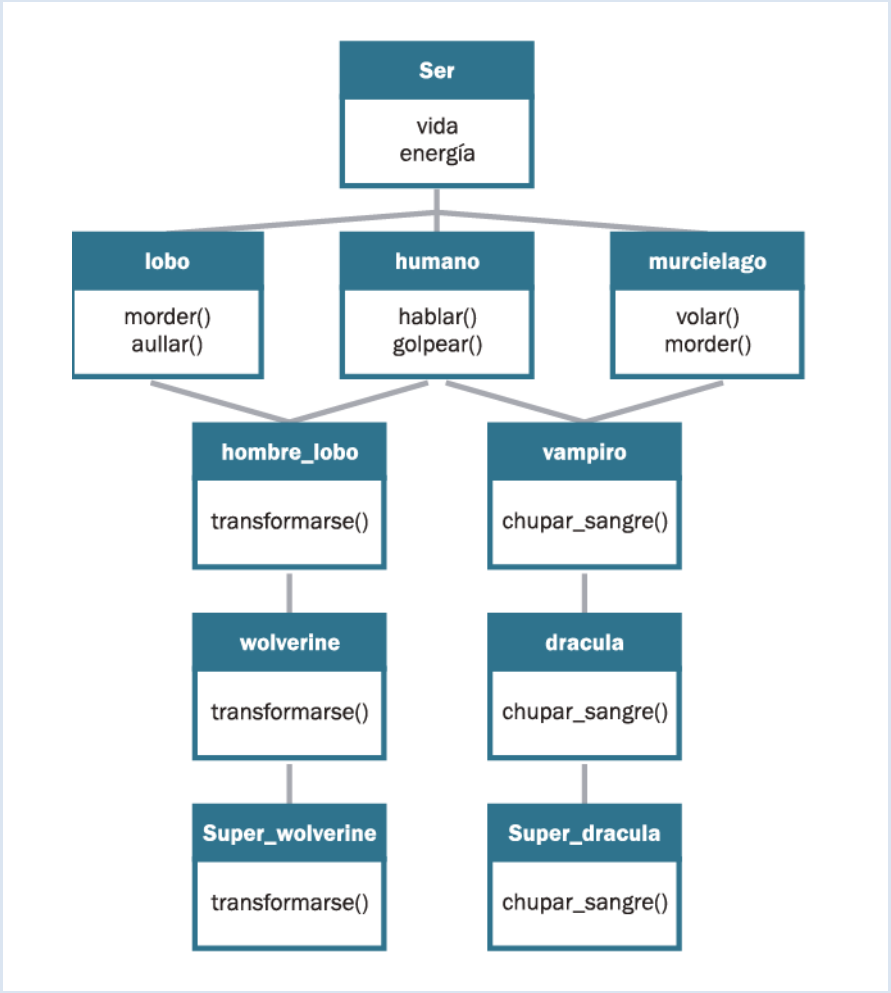

Figura 3. *Diseño de clases.*

```
#include<iostream>
#include <ctime>
#include <cstdlib>
#include <iomanip>

using namespace std;


/*****************************************************/
// Clases de 1er nivel
/*****************************************************/

/***************SER*****************/
class ser {
public:
    ser();      // constructor
    int tirar_dado();
    void recibir_dano(int);     //recibir danio
    bool muerto();              //true si esta muerto, false sino
    int ver_energia();          // el get de energia
    int ver_vida();             //el get de vida

protected:
    int vida;
```

```
    int energia;


};
ser::ser(){
    energia=20;
    vida=40;
}
int ser::tirar_dado(){
    srand (time(NULL));
    cout<<"tiraste dado"<<endl;
    return (rand()%6 +1);
}

int ser::ver_energia(){return energia;}
int ser::ver_vida(){return vida;}
void ser::recibir_dano(int a){
    vida-= a;

}

bool ser::muerto(){
    return (vida>0)? false: true;
}

/*************************************/

/*****************************************************/
// Clases de 2do nivel
/*****************************************************/

/****************LOBO*******************/
class lobo: virtual public ser{
public:
    virtual int morder();           // saca 5 de vida al oponente, consume 5
de energia
    void aullar();                  // recupera 2 de energia

};
void lobo::aullar(){
    cout<<endl<<"AUUUUUUUUUUUUUUUUUUUU"<<endl<<"Recuperaste 2 de
energia!"<<endl;
    energia+=2;
}
int lobo::morder(){
    if(energia>6){
        energia-=6;

        cout<<endl<<"Mordiste! Sacaste 5 a tu oponente, te quedan
"<<energia<<" de energia"<<endl;
        return 5;
    }
    else {
        int dado=tirar_dado();
        cout<<endl<<"tiro de dado:"<<dado<<endl;
        if(dado*energia >12){
            energia=0;
            cout<<endl<<"Mordiste! Sacaste 5 a tu oponente, te quedan
"<<energia<<" de energia"<<endl;
            return 5;
        }
        cout<<endl<<"No pudiste morder"<<endl;
        return 0;
    }

    return 0;
}

/*************************************/
```

```
/***************MURCIELAGO******************/
class murcielago: virtual public ser{

public:

    void volar();                       //recupera 2 de energia
    virtual int morder();               //saca 5 de vida al oponente, ocupa
5 de energia
};
int murcielago::morder(){
    if(energia>6){
        energia-=6;
        cout<<endl<<"Mordiste! Sacaste 5 a tu oponente, te quedan
"<<energia<<" de energia"<<endl;
        return 5;
    }
    else {
        int dado=tirar_dado();
        cout<<endl<<"tiro de dado:"<<dado<<endl;
        if(dado*energia >12){
            energia=0;
            cout<<endl<<"Mordiste! Sacaste 5 a tu oponente, te quedan
"<<energia<<" de energia"<<endl;
            return 5;
        }
        cout<<endl<<"No pudiste morder"<<endl;
        return 0;
    }

    return 0;
}

void murcielago::volar(){
    cout<<endl<<"Te escapaste volando"<<endl<<"Recuperaste 2 de
energia!"<<endl;
    energia+=2;
}


/*********************************/


/***************HUMANO*************/
class humano: virtual public ser{
public:
    int pegar();                // saca 2 de vida al oponente, consume 3 de
energia
    void hablar();              // recupera 3 de energia
};
int humano::pegar(){
    if(energia>3){
        energia-=3;
        cout<<endl<<"Pegaste! Sacaste 2 a tu oponente, te quedan
"<<energia<<" de energia"<<endl;
        return 2;
    }
    else {
        int dado=tirar_dado();
        cout<<endl<<"tiro de dado:"<<dado<<endl;
        if(dado*energia >5){
            energia=0;
            cout<<endl<<"Pegaste! Sacaste 2 a tu oponente, te quedan
"<<energia<<" de energia"<<endl;
            return 2;
        }
        cout<<endl<<"No pudiste pegar"<<endl;
        return 0;
    }

    return 0;
```

```
}

void humano::hablar(){
    cout<<endl<<"Bla Bla Bla Bla"<<endl<<"Recuperaste 3 de energia!"<<endl;
    energia+=3;
}

/************************************************/

/******************************************************/
// Clases de 3er nivel
/******************************************************/


/*************** HOMBRE LOBO *************/
class hombre_lobo : public humano, public lobo {
public:
    virtual void transformarse(void)=0;
    virtual void menu(void)=0;
    virtual int opciones()=0;
    virtual ~hombre_lobo(void);

};

/************************************************/
hombre_lobo::~hombre_lobo(){cout<<"Muerto!"<<endl;}

/*************** VAMPIRO *************/
class vampiro : public humano, public murcielago {
public:
    virtual void chupar_sangre(void)=0;
    virtual void menu(void)=0;
    virtual int opciones()=0;
    virtual ~vampiro(void);
};

vampiro::~vampiro(){cout<<"Muerto!"<<endl;}
/************************************************/

/******************************************************/
// Clases de 4to nivel
/******************************************************/

/*************** DRACULA *************/
class dracula : public vampiro{
public:
    int opciones();         // ve la cantidad de opciones del menu (solo
para la interfaz)
    void chupar_sangre();   // metodo que sirve para transformarse
    void menu();            //menu
    ~dracula(void);         //destructor

};

int dracula::opciones(){return 5;}
void dracula::chupar_sangre(){

    cout<<"AGHHHGHHG . . ."<<endl;
    cout<<"La sangre de tu enemigo te transforma en Super Dracula, ahora tu
mordida saca el doble"<<endl;
}

void dracula::menu(){

    cout<<endl<<"Jugador Dracula elige movimiento"<<endl;
    cout<<setw(5)<<"1: morder"<<endl;
    cout<<setw(5)<<"2: pegar"<<endl;
    cout<<setw(5)<<"3: hablar"<<endl;
    cout<<setw(5)<<"4: volar"<<endl;
    cout<<setw(5)<<"5: chupar sangre"<<endl;
```

```
}
dracula::~dracula(){cout<<"Muerto!"<<endl;}
/************************************/

/******* WOLWERINE *****************/
class wolwerine : public hombre_lobo{
public:
    int opciones();         // ve la cantidad de opciones del menu (solo
para la interfaz)
    void transformarse();   // metodo que sirve para transformarse
    void menu();            //menu
    ~wolwerine(void);       //destructor
};

wolwerine::~wolwerine(){cout<<"Muerto!"<<endl;}

int wolwerine::opciones(){return 5;}
void wolwerine::transformarse(){
    cout<<"AGHHHGHHG . . ."<<endl;
    cout<<"Te transformaste en un Super Wolwerine, ahora tu mordida saca el
doble"<<endl;
}

void wolwerine::menu(){

    cout<<endl<<"Jugador Wolverine elige movimiento"<<endl;
    cout<<setw(5)<<"1: morder"<<endl;
    cout<<setw(5)<<"2: pegar"<<endl;
    cout<<setw(5)<<"3: hablar"<<endl;
    cout<<setw(5)<<"4: aullar"<<endl;
    cout<<setw(5)<<"5: transformarse"<<endl;
}

/************************************/

/******* SUPER DRACULA *****************/
class Superdracula : public dracula{
public:
    int opciones();             // muestra la cantidad de opciones (solo
para interfaz)
    Superdracula(int,int);      //constructor
    int morder();               //el mismo metodo virtual pero saca doble
    void menu();                // menu
    ~Superdracula(void);        //destructor
};


Superdracula::Superdracula(int v,int e){
    energia = e;
    vida = v;
}

Superdracula::~Superdracula(){cout<<"Muerto!"<<endl;}
int Superdracula::opciones(){return 4;}
int Superdracula::morder(){

    if(energia>6){
        energia-=6;
        cout<<"Mordiste! Sacaste 10 a tu oponente, te quedan "<<energia<<"
de energia"<<endl;
        return 10;
    }
    else {
        int dado=tirar_dado();
        cout<<endl<<"tiro de dado:"<<dado<<endl;
        if(dado*energia >15){
            energia=0;
            cout<<"Mordiste! Sacaste 10 a tu oponente, te quedan
"<<energia<<" de energia"<<endl;
            return 10;
        }
```

```
        cout<<"No pudiste morder"<<endl;
        return 0;
    }

    return 0;
}


void Superdracula::menu(){



    cout<<endl<<"Jugador Super Dracula elige movimiento"<<endl;
    cout<<setw(5)<<"1: morder"<<endl;
    cout<<setw(5)<<"2: pegar"<<endl;
    cout<<setw(5)<<"3: hablar"<<endl;
    cout<<setw(5)<<"4: volar"<<endl;


}
/*************************************/


/******* SUPER WOLWERINE *****************/
class Superwolwerine : public wolwerine {
public:
    int opciones();                 //muentra cantidad de opciones (solo
para interfaz)
    Superwolwerine(int,int);        //constructor
    int morder();                   //el mismo metodo vritual pero saca
doble
    void menu();                    //menu
    ~Superwolwerine(void);      //destructor
};
Superwolwerine::Superwolwerine(int v,int e){
    energia = e;
    vida = v;
}

Superwolwerine::~Superwolwerine(){cout<<"Muerto!"<<endl;}
int Superwolwerine::opciones(){return 4;}
int Superwolwerine::morder(){

    if(energia>6){
        energia-=6;
        cout<<"Mordiste! Sacaste 10 a tu oponente, te quedan "<<energia<<"
de energia"<<endl;
        return 10;
    }
    else {
        int dado=tirar_dado();
        cout<<endl<<"tiro de dado:"<<dado<<endl;
        if(dado*energia >12){
            energia=0;
            cout<<"Mordiste! Sacaste 10 a tu oponente, te quedan
"<<energia<<" de energia"<<endl;
            return 10;
        }
        cout<<"No pudiste morder"<<endl;
        return 0;
    }

    return 0;
}


void Superwolwerine::menu(){

    cout<<endl<<"Jugador Super Wolwerine elige movimiento"<<endl;
    cout<<setw(5)<<"1: morder"<<endl;
    cout<<setw(5)<<"2: pegar"<<endl;
    cout<<setw(5)<<"3: hablar"<<endl;
```

```
    cout<<setw(5)<<"4: aullar"<<endl;
}
/************************************/




class juego {

public:
    juego();        //constructor
    vampiro *MiVampiro;
    hombre_lobo *MiHombre_Lobo;
    char turno;
    bool ganador;   // bandera que se activa cuando hay un ganador
    char gano;      //la clase ganadora
    void jugar();   //metodo que mantiene el ciclo del juego
    void cambia_jugador();  //renueva los punteros
    ~juego(void);

private:

    void cambia_turno();    //cambia el turno del jugador
    void mostrar();         //crea una interfaz
    void mostrar_menu();    //llama a los metodos de muestra de las clases
    void ingresar();        //maneja el ingreso de datos
    void actuar(int);       //toma accion sobre la opcion elegida por el
usuario
    void busca_ganador();   //detecta si hay un ganador

};

juego::juego(){

    MiHombre_Lobo = new wolwerine();
    MiVampiro = new dracula();
    turno = 'V';
    ganador=false;
    gano = 'V';
}

juego::~juego(){
    delete MiHombre_Lobo;
    delete MiVampiro;
}

void juego::mostrar(){

    cout<<endl;
    cout<<setw(10)<<"JUGADOR:
"<<"Wolverine"<<setw(10)<<"Dracula"<<endl<<endl;
    cout<<setw(10)<<"VIDA: "<<MiHombre_Lobo-
>ver_vida()<<setw(15)<<MiVampiro->ver_vida()<<endl<<endl;
    cout<<setw(10)<<"ENERGIA: "<<MiHombre_Lobo-
>ver_energia()<<setw(15)<<MiVampiro->ver_energia()<<endl<<endl;

    cout<<endl;
}



void juego::mostrar_menu(){
    (turno == 'V') ? MiVampiro->menu() : MiHombre_Lobo->menu();
}

void juego::ingresar(){

    mostrar_menu();
    int op;
    cout<<"Elige: ";
```

```
    cin>>op;

    int vidaJ = (turno == 'V')? MiVampiro->ver_vida() : MiHombre_Lobo-
>ver_vida();
    int Cop = (turno == 'V')? MiVampiro->opciones() : MiHombre_Lobo-
>opciones();

    bool cambio=false;

    if (op == 5 && vidaJ<11 && Cop == 5){

        cambio = true;
        op=1;
    }

    while (op>4 || op<1){
        mostrar_menu();
        cout<<"Opcion incorrecta, eliga nuevamente: ";
        if (op == 5)
            cout<<"Debes tener menos de 11 de vida para transformarte en
Super"<<endl;
        cin>>op;
    }


    if (cambio)
        cambia_jugador();
    else
        actuar(op);

}



void juego::actuar(int op){

    switch(op){
    case 1:
        (turno == 'L') ? MiVampiro->recibir_dano(MiHombre_Lobo->morder()) :
MiHombre_Lobo->recibir_dano(MiVampiro->morder());break;
    case 2:
        (turno == 'L') ? MiVampiro->recibir_dano(MiHombre_Lobo->pegar()) :
MiHombre_Lobo->recibir_dano(MiVampiro->pegar());break;
    case 3:
        (turno == 'L') ? MiHombre_Lobo->hablar() : MiVampiro-
>hablar();break;
    case 4:
        (turno == 'L') ? MiHombre_Lobo->aullar() : MiVampiro->volar();break;
    case 5:
        (turno == 'L') ? MiHombre_Lobo->transformarse() : MiVampiro-
>chupar_sangre();break;
    }
}



void juego::cambia_turno(){
    turno = (turno == 'L') ? 'V' : 'L';
}


void juego::busca_ganador(){

    if (MiHombre_Lobo->muerto()){

        gano = 'V';
        ganador=true;
    }
    if (MiVampiro->muerto()){

        gano = 'L';
        ganador=true;
```

```
    }
}
void juego::cambia_jugador(){

    if (turno == 'V'){
        MiVampiro = new Superdracula(MiVampiro->ver_vida(),MiVampiro-
>ver_energia());
        cout <<" AGHHHAHAHG la sangre de tu oponente te dieron super fuerza,
ahora tu mordisco saca doble"<<endl;
    }
    else
        MiHombre_Lobo = new Superwolwerine(MiHombre_Lobo-
>ver_vida(),MiHombre_Lobo->ver_energia());
    cout <<" AGHHHAHAHG la luna te dio fuerzas, te transformaste en super,
ahora tu mordisco saca doble"<<endl;
}


void juego::jugar(){

    int cont=0;


    while(!ganador){
        mostrar();
        ingresar();
        //mostrar();
        cambia_turno();
        busca_ganador();
        cout<<endl<<endl;
        cont++;
    }
    (gano == 'V')? cout<<endl<<"Gano el Vampiro"<<endl : cout<<endl<<"Gano
el Hombre lobo"<<endl;

}


/*************MAIN**************/



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

    juego J;
    J.jugar();

    return 0;
}
```

Script 9

Si bien la estructura del ejercicio es un poco compleja, podemos ver claramente el uso de herencia múltiple y polimorfismo. En este sentido, cabe remarcar dos cuestiones interesantes:

- En las clases *lobo* y *murcielago,* se implementaron los métodos *morder* y se declararon de forma virtual. Cuando los mismos se heredaron a las clases *hombre_lobo* y *vampiro,* se conservó la virtualidad. Esto permitió que, al transformarse los luchadores en su versión *super* (la cual tiene otra versión del método *morder*), la versión del método ejecutado fuera precisamente la de las clases hijas.
- La utilización de punteros nos permitió realizar de manera muy sencilla la actualización del estado de los luchadores cuando se convierten en su versión *super;* sólo hubo que referenciar el puntero a otro nuevo objeto.

3. Ejercicios propuestos

1. Leer atentamente los scripts y probarlos, investigar con ellos nuevas posibilidades, ir de menor a mayor.

2. El juego propuesto consiste en una carrera tipo triatlón, donde cada jugador compite con un auto, un trineo y una bicicleta.

El juego se comporta de la siguiente manera.

La competencia consiste en recorrer 90 kilómetros, 30 en cada terreno: asfalto, nieve y montaña. Lo decisivo es que no hay un orden para los terrenos de competencia.

Cuando el juego comienza, cada jugador debe elegir un orden en el cual usará los vehículos; luego se dispondrán los terrenos y comenzará la carrera.

Las reglas que cada vehículo debe seguir son las siguientes:

En asfalto, el auto va a 3 kilómetros por hora, la bicicleta a 2km/h y el trineo a 1km/h.

En montaña, el auto va a 1km/h, la bicicleta a 3km/h y el trineo a 2km/h

En nieve, el auto va a 2 km, la bicicleta a 1km/h y el trineo a 3 km/h.

Sugerencias

El ejercicio consiste, principalmente, en repasar el polimorfismo. Lo que proponemos es que el lector realice una estructura de clases de modo tal que cuando el juego cambie de vehículo, lo haga por medio de punteros a clases derivadas que se vayan actualizando.

Bibliografía

POZO CORONADO, Salvador. “Curso de c++” [en línea] [C++ con Clase] http://c.conclase.net/

GIL ESPERT, Lluís y SÁNCHEZ ROMERO, Montserrat. *El C++ por la práctica. Introducción al lenguaje y su filosofía.* ISBN: 84-8301-338-X.

GARCÍA DE JALÓN, Javier; RODRÍGUEZ, José Ignacio; SARRIEGUI, José María; BRAZÁLEZ, Alfonso. *Aprenda C++ como si estuviera en primero.*

RUIZ, Diego. *C++ programación orientada a objetos.* ISBN: 987-526-216-1.