UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño y programación de videojuegos

UNL VIRTUAL

Introducción a la programación

Unidad 8
Herencia

CONTENIDOS

1. Herencia: a imagen y semejanza

Los seres humanos tenemos genes que contienen determinada información, en la cual podemos encontrar algunas similitudes con la constitución biológica, física y hasta psicológica de nuestros padres y abuelos.

El concepto de *herencia* en programación parte de esta teoría. Desde el punto de vista del paradigma, dicho concepto constituye la columna vertebral de la Programación Orientada a Objetos (POO), ya que nos permite relacionar clases con las mismas características y agruparlas de una manera muy sencilla bajo un manto trasparente de parentesco, como lo es la familia para las personas.

Concretamente, la herencia nos permite *reusar* (volver a utilizar) en otras nuevas clases atributos y métodos de clases ya creadas. Más: podemos tomar lo que consideremos necesario de otras clases, modificarlo y refinarlo a gusto, al mismo tiempo que definimos nuevas características.

Este concepto de herencia suele ser difícil de incorporar en un principio y a veces puede causar más problemas que soluciones, ya que es una forma muy rápida de propagar errores. Sin embargo, el buen uso nos puede ayudar a ahorrar horas de trabajo y nos brinda una estructura de programación bien ordenada.

Clases base, madre, abstractas y derivadas

En la unidad anterior vimos cómo crear una clase con sus respectivos atributos y métodos, la instanciamos, usamos distintos tipos de constructores e incluso aprendimos a destruirla.

Ahora veremos cómo utilizar los conceptos de herencia para sacarle más provecho a lo aprendido.

Supongamos que queremos programar el clásico *Mortal Kombat*.[1] Lo primero que deberíamos hacer es dilucidar qué objetos necesitamos desarrollar. En un vistazo rápido podríamos pensar a cada luchador como un objeto:

- Luchador 1.
- Luchador 2.
- Luchador 3.
- …
- …
- Luchador x.

Cada luchador tendría características específicas, como movimientos y texturas, a la vez que atributos comunes con todos los otros luchadores, como la energía de combate, la silueta, los golpes básicos…

Ahora, si intentáramos hacer una clase por luchador, estaríamos repitiendo muchos recursos. En cambio, si utilizáramos una clase *Luchador* que reúna todas esas características globales, sólo las implementaríamos una sola vez y, además, nos aseguraríamos de que sean siempre idénticas. Luego, creamos una clase para cada luchador, con aquellos métodos y atributos particulares del mismo, y le haríamos heredar el resto de las características de la clase Luchador.

[1] Es una saga de videojuegos de pelea lanzada en 1992 por la compañía estadounidense Midway.

El siguiente esquema nos muestra cómo queda la relación entre clases:

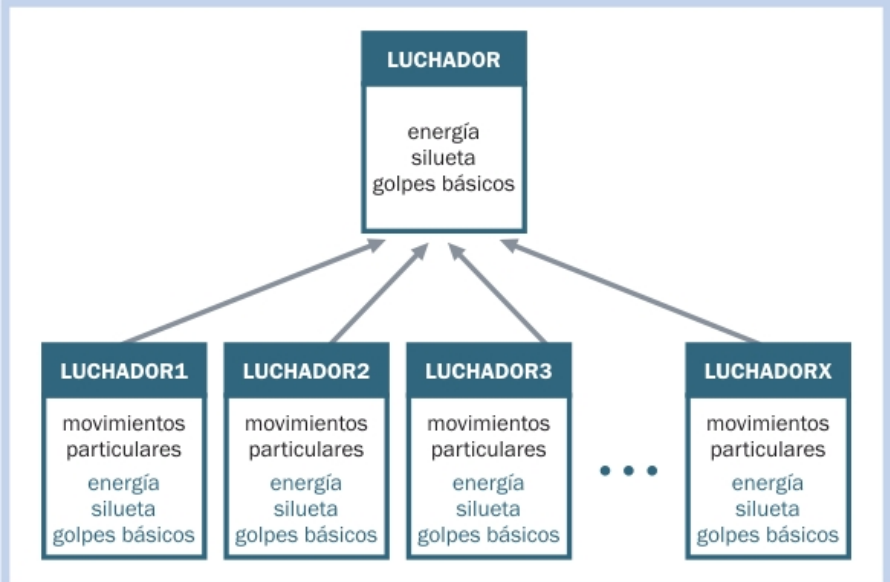

Figura 1

Tal como mencionamos, luchador1, luchador2, luchador3,… luchadorX heredan características de Luchador.

Esto implica que cuando luchador1, 2, 3,..X se implementen, cada clase tendrá a su disposición todo aquello que heredó de Luchador, tal como si fuera propio.

De la manera en que lo planteamos, la clase Luchador no tiene un explícito, es decir, no existe un luchador que sea un objeto de Luchador, sino luchadores que pertenecen a luchardor1, 2, 3, etc. Es decir, la clase Luchador se crea sólo para ser heredada. A este tipo de clases se las denomina *clases abstractas,* porque se implementan pero no se instancian. También existen clases que se declaran, se instancian y heredan sus atributos y métodos. Las mismas son llamadas *clases base* o *clases madre.* En tanto, las que reciben lo heredado de estas clases se denominan *clases derivadas.*

Como ejemplo de cada clase podemos mencionar:

La clase luchador con sus características generales es una clase abstracta. En tanto, cada luchador con sus elementos particulares son clases derivadas.

Una clase base con características particulares, que se instancie y herede, no es quizás la más común a la hora de desarrollar videojuegos. Aquí es usual encontrarse con objetos bien personificados que, si guardan similitud con otro, seguramente ocurre porque ambos son clases derivadas de otra abstracta.

Sin embargo podríamos proponer, a modo de concepto, a *Koopa*[2] como una clase base de las demás tortugas que aparecen en el juego *Mario Bros*.[3] En la práctica es poco probable que se haya desarrollado de esta forma, dada la diversidad existente entre los personajes. Es más factible aún que los hongos y las tortugas sean clases derivadas de alguna general.

Luego de esta larga introducción conceptual, veamos cómo implementamos lo dicho en C++.

[2] *Koopa el Rey* es el malo principal de la saga *Mario Bros.* Es una tortuga gigante con caparazón espinoso, garras y cuernos.
[3] Videojuego arcade desarrollado por Nintendo en 1983.

Partamos del ejemplo de la clase abstracta Luchador:

```
1   class Luchador {
2   // atributos
3   int energia;
4
5   //metodos
6   void golpear(int btn);
7   };
```

Script 1

El atributo *energia* y el método *golpear* van a ser usados en todas las clases derivadas, por lo que no hace falta implementarlos, aunque sí se hará lo propio con los métodos y atributos particulares de la clase Luchador.

Implementemos una clase *Luchador1,* que es derivada de Luchador y tiene sus propias características, además de las heredadas…

Para conseguir la herencia, la clase derivada se declara de la misma forma que veníamos usando, pero colocando el operador *dos puntos* (:) luego del nombre y un modificador de acceso a la clase madre.

Primero, veamos el concepto con el modificador *public* y luego investiguemos mejor qué efecto tienen los otros modificadores:

```
1   class Luchador1 : public Luchador {
2
3   // atributos
4   int tipo_golpe;
5
6   //metodos
7   void golpe_especial(int btn);
8
9   };
```

Script 2

Veamos cómo queda nuestro ejemplo final, accediendo a los métodos:

```
1   #include <cstdlib>
2   #include <iostream>
3
4   using namespace std;
5
6   /****   DECLARACION LUCHADOR    *****/
7
8   class Luchador {
9
10  private:
11  // atributos
12    int energia;
13
14  public:
15  //metodos
16    void golpear(int btn);
17  };
18
19
20  /****  DECLARACION LUCHADOR 1   *****/
21
22  class Luchador1 : public Luchador {
23
24  private:
25
26  // atributos
27    int tipo_golpe;
```

```
28
29  public:
30
31  //metodos
32    void golpe_especial(int btn);
33
34  };
35
36
37  // *** IMPLEMENTACION *** //
38
39
40  //* metodos de la clase abstracata *//
41
42  void Luchador::golpear(int bt){
43
44    cout <<"Metodo golpear"<<endl;
45  }
46
47  //* metodos de la clase particular Luchador1 *//
48
49  void Luchador1::golpe_especial(int bt){
50
51    cout <<"Metodo golpe especial"<<endl;
52  }
53
54
55  // *** MAIN *** //
56
57  int main(int argc, char *argv[])
58  {
59
60  Luchador1 Bajo_Cero;
61  Bajo_Cero.golpear(1);
62  Bajo_Cero.golpe_especial(1);
63
64  return 1;
65  }
```

Script 3

Aquí modificamos el acceso a los métodos de ambas clases, haciéndolos públicos, para que se pueda llegar a ellos desde el exterior, tal como vimos en la unidad anterior cuando investigamos sobre el control de acceso a los métodos.

En el main creamos un Luchador1 llamado *Bajo_Cero.* El mismo tiene los métodos golpear y *golpe_especial.* El primero es heredado, pero esto es trasparente para el usuario.

Con esto, entonces, cerramos el concepto de herencia en C++. Ahora nos queda por ver cómo son los modificadores de acceso para las clases derivadas y qué ocurre con los constructores.

Modificadores de acceso en herencia

Al igual que en los métodos, en la derivación de la herencia también existen modificadores de acceso que nos permiten controlar qué clases podrán heredar los atributos y métodos de la clase madre.

Antes de introducirnos en el tema, vamos a retomar un concepto pendiente de la unidad anterior...

Dijimos que para los métodos y atributos de una clase podemos definir tres tipos de modificadores de acceso: *private, public* y *protected*, aunque a este último sólo lo mencionamos con la promesa de abordarlo cuando tuviéramos los conocimientos

necesarios para su utilización. Pues bien, ahora que vimos herencia estamos capacitados para comprender el modificador *protected.*

Cuando algún atributo o método es declarado como protected, este sólo puede ser accedido desde la misma clase o clases derivadas pero *no* desde el exterior u otra clase externa. Podríamos resumir la idea diciendo que *el modificador de acceso protected es público para adentro y privado para afuera.*

Veamos un ejemplo con el mismo código del *Script 3,* pero esta vez designando como protected al atributo energia de la clase abstracta Luchador y accediendo al mismo desde la propia clase (derivada), intentándolo (fallidamente) desde el exterior.

```
1   #include <cstdlib>
2   #include <iostream>
3
4   using namespace std;
5
6   /****     DECLARACION       *****/
7
8   class Luchador {
9
10  protected:
11  // atributos
12    int energia;
13
14  public:
15  //metodos
16    void golpear(int btn);
17  };
18
19
20  class Luchador1 : public Luchador {
21
22
23  private:
24
25  // atributos
26    int tipo_golpe;
27
28  public:
29  //metodos
30    void golpe_especial(int btn);
31
32  };
33
34  // *** IMPLEMENTACION *** //


35  //* metodos de la clase abstracata *//
36
37  void Luchador::golpear(int bt){
38
39  energia = bt;  // dentro de la misma clase
40
41  cout<<"Metodo golpear:"<<energia<<endl;
42  }
43
44  //* metodos de la clase particular Luchador1 *//
45
46  void Luchador1::golpe_especial(int bt){
47
48    energia = bt; // dentro de la clase derivada
49
50    cout<<"Metodo golpe especial:"<<energia<<endl;
51  }
52
53
54  int main(int argc, char *argv[])
55  {
56
57  Luchador1 Bajo_Cero;
```

```
58  Bajo_Cero.golpear(1);
59  Bajo_Cero.golpe_especial(1);
60
61  // No puedo acceder ya que esta protegido
62
63  // Bajo_Cero.energia = 1;
64
65  return 1;
66  }
```

Script 4

Ahora que energia de la clase Luchador es protected, sólo podrá ser modificarlo desde la misma clase o desde Luchador1, que es su derivada. Si descomentamos la línea 63 e intentamos compilar, recibiremos un mensaje de error, avisando que ese recurso está protegido.

Como ya sabemos con claridad el efecto de cada modificador de acceso, estamos listos para ver cómo se usan en la herencia.

En el *Script 3,* en la línea 22, declaramos a la clase Luchador como public de Luchador1 y no cuestionamos por qué. Es como si hubiésemos dicho implícitamente que para heredar elementos de una clase, debemos declararla pública para su clase derivada. Pero esto no es así. Como mencionamos anteriormente, los modificadores de acceso que se utilizan para manejar los permisos al interior de la clase pueden darnos la misma riqueza a la hora de heredar. Los modificadores que podemos usar son los mismos: public, private y protected.

El efecto que ellos tienen sobre las clases derivadas son los siguientes:

public: todos los métodos conservan sus modificadores de acceso para la clase derivada. Los elementos públicos son públicos, los protegidos, protegidos y los privados, obviamente, no se heredan.

protected: los métodos públicos y protegidos se heredan como protegidos, es decir, pueden usarse dentro de la clase derivada, pero no pueden ser accedidos desde el exterior. Los privados siguen siendo así, o sea, no se heredan.

private: todos los métodos se heredan como privados, salvo aquellos que no se heredan.

La siguiente tabla resume lo que acabamos de decir:

Modificador en clase base (sobre los métodos y atributos)	Modificador utilizado en la herencia	Modificador resultante en la clase derivada
public		public
protected	public	protected
private		*no se hereda*
public		protected
protected	protected	protected
private		*no se hereda*
public		private
protected	private	private
Private		*no se hereda*

Ahora, visualicémoslos en un ejemplo:

Tomaremos la clase Luchador, que seguiremos usando como clase base, e implementaremos tres clases más: L*uchador1, Luchador2 y Luchador3,* cada una con un modificador distinto (public, protected y private, respectivamente). A su vez,

sólo a manera de ejemplo, le sumaremos dos métodos más a la clase madre: *patear* y *saltar*, de forma que quedarán de la siguiente manera:

golpear: public.

patear: protected.

saltar: private.

```
#include <cstdlib>
#include <iostream>

using namespace std;

/****     DECLARACION       *****/

class Luchador {

//metodos
public: void golpear(int btn);
protected: void patear(int btn);
private: void saltar(int btn);

};

/***** clases derivadas **********/

class Luchador1 : public Luchador {
  public: void hace_todo();};

class Luchador2 : protected Luchador {
  public: void hace_todo();};

class Luchador3 : private Luchador {
  public: void hace_todo();};


// *** IMPLEMENTACION *** //

//* metodos de la clase abstracta *//

void Luchador::golpear(int bt){
  cout <<"golpear: "<<bt<<endl;
}
void Luchador::patear(int bt){
  cout <<"patear: "<<bt<<endl;
}

void Luchador::saltar(int bt){
  cout <<"saltar: "<<bt<<endl;
}

//* metodos de las clases derivadas  *//

void Luchador1::hace_todo(){
  cout <<"Luchador1 Hace todo: "<<endl;
  golpear(1);
  patear(1);
}

void Luchador2::hace_todo(){

  cout <<"Luchador2 Hace todo: "<<endl;
  golpear(1);
  patear(1);
}

void Luchador3::hace_todo(){
  cout <<"Luchador3 Hace todo: "<<endl;
  golpear(1);
  patear(1);
}
```

```
64
65  int main(int argc, char *argv[])
66  {
67
68    Luchador1 Bajo_Cero;
69    Luchador2 Alacran;
70    Luchador3 Chin_Chan;
71
72  /* Bajo_Cero hereda como publico */
73
74  Bajo_Cero.golpear(1);
75  // Bajo_Cero.patear(1); // Esta protegido
76  // Bajo_Cero.saltar(1); // No esta heredado
77
78  //    Alacran.golpear(1);  // Esta protegido
79  //    Alacran.patear(1);   // Esta protegido
80  //    Alacran.saltar(1);   // No se hereda
81
82  //    Chin_Chan.golpear(1);   // Es privado
83  //    Chin_Chan.patear(1);    // Es privado
84  //    Chin_Chan.saltar(1);    // No se hereda
85
86  Bajo_Cero.hace_todo();
87  Alacran.hace_todo();
88  Chin_Chan.hace_todo();
89
90  return 1;
91  }
```

Script 5

A las clases Luchador1, Luchador2 y Luchador3 se les agregó el método público *hace_todo,* que llama a los métodos internos para verificar que efectivamente, de manera local, la clase tiene acceso a los métodos heredados.

De la línea 75 a la 84, todas están comentadas. Esto se debe a que no tenemos acceso a esos métodos, pero si vamos descomentando de a una línea, iremos viendo los errores que nos arroja el compilador: “imposible acceder al método” (para aquellos heredados sin permiso externo) y “método privado” (para aquellos que no fueron heredados porque fueron declarados como privados en la clase madre).

El único objeto que pudo convocar un método fue Bajo_Cero, que instanció a golpear, ya que la clase que instancia Bajo_Cero (Luchador1) fue heredada de manera pública. A su vez, como el método es público en la clase madre, se puede acceder al mismo desde la interfaz de la clase.

De esta forma, si Bajo_Cero lo desea, puede golpear, pero si *Chin_Chan* o *Alacran* quieren golpear a alguien, sólo lo van a poder hacer desde el método hace_todo, lo cual implica que, además de golpear, tendrán que patear. Aquí se ve claramente la importancia de tener un buen manejo de los controles de acceso. Con una correcta aplicación de las etiquetas de acceso obligamos a todos los objetos a acceder a los métodos privados por un único método público (hace_todo), el cual –en este caso– no hace más que llamar a los métodos protegidos y privados, si bien podríamos haber puesto algún verificador de variables, un controlador de condiciones o un detector de errores previamente a la ejecución del método.

Lo más importante de la herencia es precisamente este punto. Notemos cómo con una sola clase madre pudimos crear distintas clases derivadas, controlando los modificadores de acceso, tanto de los métodos internos como de la herencia.

En los scripts anteriores mostramos ejemplos con un solo nivel de herencia. Sin embargo, podría darse el caso de una clase derivada de la clase derivada. Sea cual sea el nivel de derivación, las reglas de control de acceso se mantienen. Si partimos de una clase base que tiene un método público y su clase derivada lo

hereda como protegido, la clase hija de la derivada lo verá como protegido y desconocerá si alguna vez tuvo otra permisividad.

Este juego tiene algunos secretos. Si jugamos con Bajo_Cero y superamos cierta cantidad de puntos, podemos desbloquear al personaje *Ultra_Bajo_Cero,* que ejecuta los mismos métodos golpear y patear de la clase base, con la diferencia de que hace el doble de daño. El cálculo del golpe sigue siendo el mismo, pero ahora el efecto se duplica, si bien esto debe ser trasparente para la interfaz. Este objeto se instancia a partir de una nueva clase derivada de Luchador1.

Debemos sortear, entonces, dos problemas: primero, cómo serán los accesos en la herencia, y segundo, cómo hacemos para ejecutar un método heredado, que efectúe una acción diversa.

Para solucionar el primer problema debemos mantener en público el acceso de la clase Luchador1 para que los métodos heredados puedan seguir siendo visibles desde el exterior. En tanto, en la clase derivada de Luchador1, a la cual llamaremos *Luchador1_1,* la herencia debe ser privada o protegida porque estos métodos no pueden ser visibles desde la interfaz.

Para solucionar el segundo problema debemos sobrescribir los métodos golpear y patear de la clase Luchador1_1. Estos métodos sólo invocarán a los métodos originales, pero con el parámetro multiplicado por *2* (para ejecutar el doble de daño).

Cuando los métodos nuevos llamen a los métodos heredados que llevan el mismo nombre, deberán hacerlo de manera explícita. De lo contrario, el compilador no reconocerá a cuál de los dos métodos se está haciendo referencia y podrá colgar el programa.

```
1   #include <cstdlib>
2   #include <iostream>
3
4   using namespace std;
5
6   /****     DECLARACION       *****/
7
8   class Luchador {
9
10          //metodos
11    public:
12
13         void golpear(int e);
14         void patear(int e);
15  };
16
17  /***** clases derivadas **********/
18
19  class Luchador1 : public Luchador {};
20
21  class Luchador1_1 : private Luchador1 {
22    public:
23       void golpear(int e);
24       void patear(int e);
25  };
26
27
28  // *** IMPLEMENTACION *** //
29
30
31  //* metodos de la clase abstracata *//
32
33  void Luchador::golpear(int bt){
34      cout <<"golpear: "<<bt<<endl<<endl;
35  }
36  void Luchador::patear(int bt){
```

```
37      cout <<"patear: "<<bt<<endl<<endl;
38  }
39
40  //* metodos de las clases derivadas  *//
41
42  void Luchador1_1::golpear(int bt){
43      // llamada explicita
44      cout<<"golpe doble"<<endl;
45
46      Luchador::golpear(bt*2);
47  }
48
49  void Luchador1_1::patear(int bt){
50     // llamada explicita
51      cout<<"patada doble"<<endl;
52
53      Luchador::patear(bt*2);
54  }
55
56
57  int main(int argc, char *argv[])
58  {
59
60    Luchador1 Bajo_Cero;
61    Luchador1_1 Ultra_Bajo_Cero;
62
63    Bajo_Cero.golpear(2);
64    Bajo_Cero.patear(2);
65
66    Ultra_Bajo_Cero.golpear(2);
67    Ultra_Bajo_Cero.patear(2);
68
69  return 1;
70  }
```

Script 6

La sobreescritura que realiza la clase Luchador1_1 nos sirvió para ejecutar los mismos métodos sin necesidad de cambiar de nombre. En la práctica, esta técnica se utiliza para ocultar los métodos de las clases superiores.

La ejecución explícita en las líneas 46 y 53 fue posible sólo porque la clase Luchador 1_1 tiene acceso a esos métodos.

Constructores y destructores en herencia

Cuando instanciamos un objeto que tiene herencia en otro, debemos mantener el orden en el cual llamamos a sus constructores, a fin de que se mantenga la coherencia a la hora de crear las dependencias.

Si una clase *A* (base) hereda a otra clase *B* (derivada), a la hora de instanciar los objetos, primero debemos llamar al constructor de A y luego, al de B.

Lo mismo ocurre con los constructores: primero debemos destruir el objeto derivado y luego, el base, de manera inversa a los constructores.

El problema surge cuando la clase base posee un constructor con parámetros. Por ejemplo, si Luchador requiere un número de nivel inicial, pero nunca instanciamos Luchador sino que sólo usamos sus métodos.

Hay dos formas de salvar esta situación:

- Si conocemos el/los parámetro/s que requiere la clase Luchador, podemos crear un constructor en Luchador1 que le pase estos parámetros. En este caso, el constructor de Luchador1 quedaría vacío.

- Otra opción es crear un constructor de Luchador1 que requiera parámetros de Luchador en su creación para luego transferirle estos parámetros.

La manera de realizar la primera opción es:

```
1   #include <cstdlib>
2   #include <iostream>
3
4   using namespace std;
5
6   /****     DECLARACION       *****/
7
8   class Luchador {
9       public:
10
11      Luchador(int a);
12  };
13
14
15  class Luchador1 : public Luchador {
16       public:
17
18           Luchador1();
19  };
20
21
22  // *** CONSTRUCTORES *** //
23
24
25  Luchador::Luchador(int a){
26
27          cout <<"Se creo Luchador: "<<a<<endl<<endl;
28  }
29
30
31  Luchador1::Luchador1() : Luchador(5){
32
33       cout <<"Se creo Luchador1: "<<endl<<endl;
34  }
35
36
37
38
39  int main(int argc, char *argv[])
40  {
41
42  Luchador1 Bajo_Cero;
43
44  return 1;
45  }
```

Script 7

Y la segunda:

```
1   #include <cstdlib>
2   #include <iostream>
3
4   using namespace std;
5
6   /****     DECLARACION       *****/
7
8   class Luchador {
9
10  public:
11       Luchador(int a);
12  };
13
14
```

```
15
16  class Luchador1 : public Luchador {
17        public:
18
19              Luchador1(int b);
20
21  };
22
23  // *** IMPLEMENTACION *** //
24
25  Luchador::Luchador(int a){
26            cout <<"Se creo Luchador: "<<a<<endl<<endl;
27  }
28
29
30  Luchador1::Luchador1(int b) : Luchador(b){
31            cout <<"Se creo Luchador1: "<<b<<endl<<endl;
32  }
33
34
35  int main(int argc, char *argv[])
36  {
37
38  Luchador1 Bajo_Cero(5);
39
40  return 1;
41  }
```

Script 8

Como vemos, en las líneas 31 del *Script 7* y 30 del *Script 8*, lo que debemos hacer es colocar los dos puntos (:) a continuación del cierre de los paréntesis del constructor. En el primer caso, el constructor de la clase derivada queda vacío, pero le pasamos los parámetros conocidos al constructor de la clase base. En la segunda opción, tomamos un parámetro y lo transferimos al constructor de la clase base.

Ejercicio integrador

Como ejercicio integrador vamos a implementar el juego del águila y la liebre. Este juego es para dos jugadores, uno de ellos utiliza al águila mientras que el otro juega con la liebre. El objetivo del juego para la liebre es escapar de las garras del águila mientras que para ésta el objetivo es atrapar a la liebre.

El juego tiene las siguientes características:

- La liebre corre hacia adelante o hacia las diagonales.
- La liebre por cada turno se mueve un casillero.
- Si la liebre llega al final del terreno sin ser agarrada, gana.
- El águila vuela hacia adelante y hacia los costados
- Cuando el águila vuela hacia adelante avanza dos casilleros.
- Cuando el águila vuela hacia los costados, no avanza (no se mueve en x, sólo en y).
- Además de estos tres movimientos ambos jugadores pueden optar por no moverse.
- Si al pasar 40 turnos no hay ganadores, el águila gana.

La forma en la cual implementamos el juego es con 4 clases:

- Animal.
- Liebre.
- Aguila.
- Juego.

La primera es una clase base que tiene algunas características comunes de ambos animales:

- Un método que mueve hacia adelante.
- Un arreglo que lleva las coordenadas del objeto.

La clase liebre tiene los siguientes componentes:

- Un método saltar que usa el método heredado de la clase madre.
- Un método que permite desplazarse hacia las diagonales.
- Un método de interfaz.

La clase águila tiene los siguientes componentes:

- Un método volar que usa el método heredado de la clase madre.
- Un método que permite desplazarse hacia los laterales.
- Un método de interfaz.

La clase juego tiene los métodos y propiedades que permiten recrear el escenario, además tiene un objeto águila y un objeto liebre.

```
#include <cstdlib>
#include <stdlib.h>
#include <iostream>
#include <iomanip>

using namespace std;

/****     CLASES       *****/

/********** Clase madre animal **********/

class animal {


public:
    int coordenadas[2]; // lleva las coordenadas del objeto

protected:

    void avanzar(int); // un metodo que ambas clases derivadas tiene en
comun
};


/********** Clase aguila, hereda animal **********/

class aguila : public animal {


public:
    aguila();           // constructor
    void volar();          // llama al metodo avanzar, pero avanza 2
casilleros
    void costado(int);  // se mueve a los laterales segun el parametro
    void menu();        // muestras las opciones posibles del objeto


};

/********** Clase liebre, hereda animal **********/

class liebre : public animal {

public:
    liebre();           // constructor
    void saltar();      // llama al metodo avanzar
    void diagonal(int); //se mueve en diagonal segun el parametro
    void menu();        //// muestras las opciones posibles del objeto
};
```

```
/********** Clase juego, es la que mantiene la llamada a los metodos
contiene un objeto liebre y un objeto aguila
**********/

class juego {

public:
    juego();        //constructor
    aguila A;       // objeto aguila
    liebre L;       //objeto liebre

    bool ganador;   // bandera que se activa cuando hay un ganador
    char gano;      //la clase ganadora
    void start();   //metodo que mantiene el ciclo del juego
    int gano_liebre();  //metodo de interfaz cuando gana la liebre
    int gano_aguila();  //metodo de interfaz cuando gana el aguila

private:
    char terreno[7][20];    //Terreno de juego
    char turno;             //turno del jugador de turno

    void cambia_turno();    //cambia el turno del jugador
    void arma_terreno();    //arma el terreno por unica vez
    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

};

/************ IMPLEMENTACIONES ******************/
/************************************************/
/************************************************/
/************************************************/


/************************************************/
/*             CLASE ANIMAL                     */
/************************************************/

/* metodo avanzar compartido por las cases derivadas */

void animal::avanzar(int a){

    coordenadas[1]+=a;
}


/************************************************/
/*             CLASE LIEBRE                     */
/************************************************/

/* constructor */

liebre::liebre(){
    coordenadas[0]=3;
    coordenadas[1]=5;
}

/* muestra el menu de las opciones */

void liebre::menu(){
    cout<<endl<<"Jugador Liebre elige movimiento"<<endl;
    cout<<setw(5)<<"1: saltar"<<endl;
    cout<<setw(5)<<"2: diagonal abajo"<<endl;
    cout<<setw(5)<<"3: diagonal arriba"<<endl;
    cout<<setw(5)<<"4: pasar turno"<<endl;
```

```
}

/* corre las coordendas hacia las diagonales */

void liebre::diagonal(int a){

    if (a == 1){
        if (coordenadas[0]<5)
            coordenadas[0]++;


    }
    else{
        if (coordenadas[0]>1)
            coordenadas[0]--;

    }

    coordenadas[1]++;

}

/* usa el metodo de la case base, avanza 1 */

void liebre::saltar(){
    avanzar(1);
}


/***********************************************/
/*             CLASE AGUILA                    */
/***********************************************/


/* constructor */

aguila::aguila(){
    coordenadas[0]=3;
    coordenadas[1]=2;
}

/* muestra el menu de las opciones */

void aguila::menu(){
    cout<<endl<<"Jugador Aguila elige movimiento"<<endl;
    cout<<setw(5)<<"1: volar"<<endl;
    cout<<setw(5)<<"2: abajo"<<endl;
    cout<<setw(5)<<"3: arriba"<<endl;
    cout<<setw(5)<<"4: pasar turno"<<endl;

}

/* corre las coordendas hacia los laterales */
void aguila::costado(int a){


    if (a == 1){
        if (coordenadas[0]<5)
            coordenadas[0]++;


    }
    else{
        if (coordenadas[0]>1)
            coordenadas[0]--;

    }

}

/* usa el metodo de la case base, avanza 2 */
```

```
void aguila::volar(){
    avanzar(2);
}

/***********************************************/
/*             CLASE JUEGO                     */
/***********************************************/

/* constructor */

juego::juego(){
    arma_terreno();
    turno = 'L';
    ganador=false;
    gano = 'A';
}

/* arma terreno por unica vez */

void juego::arma_terreno(){


    for(int i=0;i<20;i++){
        terreno[0][i]='=';
        terreno[6][i]='=';
    }

    for(int i=0;i<7;i++){
        terreno[i][0] ='|';
        terreno[i][19]='|';
    }

    for (int i=1;i<6;i++){
        for (int j=1;j<19;j++){
            terreno[i][j]=' ';
        }
    }



    terreno[A.coordenadas[0]][A.coordenadas[1]]='A';
    terreno[L.coordenadas[0]][L.coordenadas[1]]='L';
    }


/* crea la interfaz grafica */

void juego::mostrar(){

    for (int i=1;i<6;i++){
        for(int k=1;k<19;k++){
            terreno[i][k]=' ';
        }
    }

    terreno[A.coordenadas[0]][A.coordenadas[1]]='A';
    terreno[L.coordenadas[0]][L.coordenadas[1]]='L';

    for (int i=0;i<7;i++){
        for(int k=0;k<20;k++){
            cout<<setw(2)<<terreno[i][k];
        }
        cout<<endl<<endl;;

    }




}
```

```
/* muestra el menu de los objetos */

void juego::mostrar_menu(){
    (turno == 'L') ? L.menu() : A.menu();
}

/* controla el ingreso de datos del usuario */

void juego::ingresar(){

    mostrar_menu();
    int op;
    cout<<"Elige: ";
    cin>>op;

    while (op>4 || op<1){
        mostrar_menu();
        cout<<"Opcion incorrecta, eliga nuevamente: ";
        cin>>op;
    }

    actuar(op);

}

/* toma aciones segun la opcion elegida por el usuario */

void juego::actuar(int op){

    switch(op){
    case 1:
        (turno == 'L') ? L.saltar() : A.volar();break;
    case 2:
        (turno == 'L') ? L.diagonal(1) : A.costado(1);break;
    case 3:
        (turno == 'L') ? L.diagonal(2) : A.costado(2);break;
    case 4:
        (turno == 'L') ? turno = 'L' : turno = 'A';break;
    }
}

/* cambia el turno del jugador*/

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

/* detecta si hay un ganador */

void juego::busca_ganador(){

    if (L.coordenadas[1]>18){

        gano = 'L';
        ganador=true;
    }

    if(L.coordenadas[0]  ==  A.coordenadas[0]  &&  L.coordenadas[1]  ==
A.coordenadas[1])
    {
        gano = 'A';
        ganador=true;
    }
}


/* muestra interfaz de la victoria de la liebre */

int juego::gano_liebre(){

    cout<<"Gano Liebre"<<endl<<endl;
    cout<<" () ()"<<endl;
```

```
    cout<<" (`.`)"<<endl;
    cout<<"(``)(``)"<<endl;

    return 1;

}

/* muestra interfaz de la victoria del aguila */

int juego::gano_aguila(){

    cout<<"LIEBRE CAPTURADA"<<endl;

    return 1;
}

/* metodo que controla el ciclo de juego */

void juego::start(){

    int cont=0;

    while((!ganador) && (cont<40)){
        mostrar();
        ingresar();
        mostrar();
        cambia_turno();
        busca_ganador();
        cout<<"Quedan: "<<30-cont<<" turnos"<<endl;
        cont++;
    }

    if(cont==30){
        cout<<endl<<"Gano aguila por pasar 30 turnos";
    }
    else{
        int a=(gano == 'L') ? gano_liebre() : gano_aguila();
    }

}



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

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

    juego J;
    J.start();


    return 1;
}
```

Script 9

Bibliografía

GIL ESPERT, Lluís y SÁNCHEZ ROMERO, Montserrat. *El C++ por la práctica. Introducción al lenguaje y su filosofía.* Barcelona, Universidad Politécnica de Cataluña, 1999. ISBN: 84-8301-338-X.

GARCÍA DE JALÓN, Javier y otros. *Aprenda C++ como si estuviera en primero.* Escuela Superior de Ingenieros de San Sebastián (Universidad de Navarra), San Sebastián, 1998.

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