UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño y programación de videojuegos

UNL VIRTUAL

Introducción a la programación

Unidad 7
Introducción a objetos

CONTENIDOS

1. POO: primeros conceptos

La Programación Orientada a Objetos (POO) es un paradigma de programación, es decir, una forma de programar, aceptada por las tecnologías del software y la comunidad de programadores y la industria del software en general. Algunos paradigmas son la programación procedural, la programación modular, la programación lógica, la programación orientada a objetos, etc. Obviamente, esta variedad no implica que uno sea mejor que otro, sino que hay ciertos paradigmas que se ajustan mejor a determinado tipo de problemas que otros.

Es probable que en la práctica se utilice más de un paradigma a la vez, pero siempre habrá uno predominante. Por ejemplo, la POO es, justamente, un paradigma que ha ido creciendo en popularidad gracias a su adaptación para la resolución de un gran número de problemas.

Es muy importante tener en cuenta, porque aquí es donde suelen generarse confusiones, lo que un paradigma representa. Si queremos programar con objetos, no debemos hacerlo por capricho ni porque ayuda a resolver cierto problema. Es fundamental reconocer el contexto y elegir con fundamentos el paradigma que se va a utilizar. Probablemente, en el 90% de los casos, la mejor elección sea la POO. Sin embargo, esto no quita que nos preguntemos al menos una vez qué modelo vamos a usar antes de empezar a codificar.

2. Objetos: Yo objeto tu objeto

El concepto de objeto en programación no difiere mucho del que tenemos en la vida cotidiana. Si miramos a nuestro alrededor, vamos a notar muchos de ellos: el teclado, el monitor, el mouse, el MODEM, el lobo marino recuerdo de Mar del Plata, etc.

En este caso, todos constituyen objetos del ámbito informático, salvo el lobo marino, y todos ellos se relacionan mediante algún conector, incluso algunos dependen de otros. Pensemos, ahora, en un programa, o mejor en un videojuego. Para que todos nos centremos en el mismo, vamos a elegir el archiconocido *Pac-Man*[1]. ¿Qué objetos notamos ahí? Podríamos listar los siguientes:

- El Señor Pac-Man.
- Los fantasmitas.
- Las pastillitas.
- Las frutitas que aparecen de vez en cuando.

Todos ellos constituyen objetos independientes que se relacionan de alguna forma: el Pac-Man come las pastillitas; los fantasmitas comen al Pac-Man, salvo cuando éste agarra la pastilla grande, etc. Todos los objetos se relacionan entre sí. Esta es la base de la POO: pueden existir objetos aislados en un concepto teórico, pero en la práctica casi no se ven.

Para resumir, los objetos constituyen pequeñas partes de un programa que se relacionan para llevar adelante un objetivo. En el caso del Pac-Man, es el de constituir el juego tal como es. En esta primera sección veremos en detalle cuáles son los componentes de un objeto y cómo se declaran en C++.

[1] Videojuego arcade desarrollado por la compañía Namco a principio de los años 1980

Componentes de un objeto

Un objeto tiene algunos componentes bien distinguibles: sus *propiedades* y sus *métodos.*

Vamos a desarrollar estos conceptos, tomando como ejemplo al *Pac-Man,* y particularmente, como sujeto de desarrollo, al fantasmita.

El primer componente de un objeto, entonces, son las propiedades, que definen las características de un objeto. En nuestro fantasmita, las propiedades podrían ser el color, la velocidad con la que se mueve, su posición en la pantalla, el estado (si está parpadeando o no). Por ejemplo, en un televisor, las propiedades serían el ancho, el alto, el material, el color, las pulgadas, una marca.

La otra componente de los objetos son los métodos, que detallan las funciones que puede desarrollar el objeto. Así, en el fantasmita, podríamos encontrar los siguientes métodos: comer (al pacman), moverse y parpadear. En el televisor, tenemos métodos como encender, apagar, cambiar de canal, de volumen, etc.

En simples palabras, un objeto es:

objeto = propiedades + métodos

Objetos y clases en C++

Una clase es una estructura de datos. Su funcionalidad es la de crear nuevos datos para ser utilizados, muy similar al *struc*.

Veamos, por ejemplo, cómo haríamos si quisiéramos implementar el fantasmita del *Pac-Man* mediante una struct:

```
1   struct fantasmita{
2
3   char *color;
4   int estado; // 1 titilando 0 apagado
5
6   };
```

Script 1

De manera que si quisiéramos crear un fantasmita, haríamos:

```
1   int main(int argc, char *argv[])
2   {
3
4   fantasmita Rojo;
5
6   return 0;
7   }
```

Script 2

Para definirle un color y el estado, creamos dos funciones:

```
1   void define_color(fantasmita &f, char *c){
2   f.color = c;
3   }
4
5   void define_estado(fantasmita &f, int e){
```

```
6   f.estado = e;
7   }
```

Script 3

De esta manera, para definir el color y cambiar el estado, simplemente llamamos a las funciones y pasamos el fantasmita por parámetro. Pero el gran problema aquí es la permisividad.

En esta forma de declarar los fantasmitas, cualquiera puede acceder a las propiedades de la estructura. Don Pac-Man estaría encantado de poder cambiarle el estado a los fantasmitas, llamando a la función *define_estado* en cualquier momento, sin necesidad de comer la pastilla grande. Sin embargo, como programadores y muy a pesar del Señor Pac-Man, esto no es lo ideal.

La *clase* tiene dos propiedades por sobre el struct, que nos permiten controlar esto.

Primero, una clase permite tener funciones asociadas a ellas, de manera que *define_color* y *define_estado* pueden pertenecer a la clase *fantasmita* y llamarse dentro de ella. Segundo, la clase permite controlar los permisos de quienes pueden acceder a sus funciones y sus características.

Las funciones dentro de las clases se llaman *métodos* y las características se llaman *atributos.*

Para definir una clase en C++ utilizamos la siguiente sintaxis:

```
1   class fantasmita{
2
3   char *color;
4   int estado; // 1 titilando 0 apagado
5
6   };
```

Script 4

Como vemos, la sintaxis es idéntica a la del struct, salvo que la palabra clave es *class.* Para crearla desde el main, utilizamos la forma utilizada en el *Script 2*.

En esta etapa, la declaración de una clase se denomina *instanciar.* Cuando instanciamos la clase en el main, lo que obtenemos es un *objeto*. Puede parecer el mismo concepto, pero una clase es una estructura de datos sin vida, mientras que el objeto surge cuando instanciamos esa estructura y la hacemos correr en la ejecución.

Métodos y atributos

Dijimos que los atributos son aquellas características pertenecientes a las clases y los métodos son las funciones de la misma, aunque aún no explicamos cómo implementar un método en una clase en C++.

Los métodos tienen la siguiente sintaxis:

```
tipo_retorno nombre_clase::nombre_método(parámetros){
        ….
}
```

Por ejemplo, si quisiéramos agregarle los métodos *define_color* y *define_estado* a nuestra clase fantasmita, lo tendríamos que hacer así:

```
1   class fantasmita{
2
```

```
3   // atributos
4
5   char *color;
6   int estado; // 1 titilando 0 apagado
7
8   // métodos
9
10       void fantasmita::define_color(char *c){
11
12              color=c;
13
14  }
15
16        void fantasmita::define_estado(int e){
17
18              estado = e;
19
20  }
21
22  };
```

Script 5

Luego, desde el main, tendríamos que instanciar a la clase de la siguiente manera:

```
1   int main(int argc, char *argv[])
2   {
3
4   fantasmita Rojo;
5   Rojo.define_color("rojo");
6
7   return 1;
8   }
```

Script 6

El operador *punto (.)* es el que define la llamada a un componente de la clase, tanto un atributo como un método.

El *Script 6* no está mal. Sin embargo, si lo quisiéramos correr, el compilador nos arrojaría un error diciendo “no se puede acceder al método privado *define_color*”.

Esto nos lleva a la segunda propiedad que tienen las clases por sobre los struct y que habíamos mencionado en página 3los permisos que puede brindar una clase sobre sus métodos y atributos.

Tanto los atributos como los métodos de una clase pueden tener tres tipos distintos de modificadores de acceso:

- private
- public
- protected

En español, privado, público y protegido, respectivamente.

Al tipo *protected* lo dejaremos para más adelante, ya que requiere conceptos que aún no vimos ni abordaremos en esta unidad.

El tipo *private* permite que todos los atributos y métodos definidos bajo su etiqueta sean modificados (accedidos) sólo dentro de la clase.

El tipo *public* permite que todos los atributos y métodos definidos bajo su etiqueta puedan ser accedidos tanto desde el ámbito interno a la clase, como del externo.

Si no se indica lo contrario, C++ toma por defecto a todos los métodos y atributos de una clase como *private.*

Esta propiedad de dar o denegar el permiso de acceso a atributos o métodos es una de las principales características del paradigma y se denomina *encapsulamiento.*

El objeto tiene la capacidad de encapsular aquellas propiedades y métodos que no forman parte de su interfaz y se destinan para un uso interno.

Existe una tendencia en la POO de negarle la característica de *public* a los atributos, debido a que, en general, los mismos llevan características de estado que deberían ser sólo modificables por métodos de la misma clase y no de otra. Ahora, si quisiéramos definir como público el atributo *estado,* podríamos hacer en el main:

```
Fantasmita Rojo;

        Rojo.estado = 1;
```

Sin embargo, esto no está muy bien visto a nivel de seguridad. En este ejemplo se puede ver su deficiencia, ya que ese atributo podría ser modificado por cualquiera, incluso por el Sr Pac-Man, y el juego no tendría sentido. De todos modos, como aún estamos aprendiendo, pueden permitirse pasar por alto este detalle y probar cómo se accede a un atributo público.

Continuamos, ahora, con nuestro ejemplo del fantasmita.

La clase *fantasmita* tiene dos métodos y dos atributos. Lo más correcto sería definir ambos atributos como privados y ambos métodos como públicos, de manera que los atributos sólo sean modificables por el método de clase.

Esto se hace de la siguiente manera:

```
1   class fantasmita{
2
3   private:
4
5            char *color;
6             int estado; // 1 titilando 0 apagado
7
8   public:
9
10               void fantasmita::define_color(char *c){
11
12                         color=c;
13
14               }
15
16              void fantasmita::define_estado(int e){
17
18                        estado = e;
19
20              }
21
22  };
```

Script 7

Ahora nos quedaron los atributos *color* y *estado* declarados como privados. Si bien esta etiqueta está demás (ya que si no estuviera C++ los tomaría por defecto como privados), por una cuestión de organización siempre conviene agregarla. Más abajo, todos los métodos quedaron definidos como públicos. Por lo tanto, ya podemos ejecutar el *Script 6* sin problemas.

En este ejemplo tenemos sólo dos métodos y no es difícil encontrarlos bajo la etiqueta *public.* Sin embargo, si tuviéramos 100 métodos junto con algunos atributos, sería bastante engorroso andar buscando línea por línea la declaración de los mismos.

Por una cuestión organizativa, C++ permite definir las cabeceras de las clases con la declaración de los métodos y luego implementarlos aparte. De esta manera, si nosotros no conociéramos y quisiéramos saber qué métodos públicos tiene la clase *fantasmita,* podríamos fijarnos directamente en la cabecera de una forma más ordenada.

Esto tiene que ver también con una cuestión de *linkeo* de archivos que utilizan los compiladores. Más adelante veremos cómo se organizan, pero por el momento dejaremos todo en el mismo archivo y empezaremos a utilizar las cabeceras separadas de las implementaciones.

Como ejemplo, la clase *fantasmita*:

```
1   #include <cstdlib>
2   #include <iostream>
3
4   using namespace std;
5
6   // empieza declaracion de clase
7
8   class fantasmita{
9
10  private:
11
12          char *color;
13          int estado; // 1 titilando 0 apagado
14
15  public:
16           void define_color(char *c);
17           void define_estado(int e);
18  };
19
20  // termina declaracion de clase
21
22  // empieza definición de metodos
23
24  void fantasmita::define_color(char *c){
25
26           color=c;
27           cout <<"El color es "<<color<<endl;
28  }
29
30  void fantasmita::define_estado(int e){
31
32             estado = e;
33
34  }
35
36  // termina definición de metodos
37
38  int main(int argc, char *argv[])
39  {
40
41       fantasmita Rojo;
42       Rojo.define_color("rojo");
43      // Rojo.estado=0; // no puedo acceder ya que el atributo es
    privado
44
45  return 1;
46  }
```

Script 8

Notemos dónde empieza y termina cada llave. Con esta manera de organizar la clase, una persona que no la conoce mirará directamente la declaración y sin necesidad de saber cómo está implementado cada método, podrá ver que existen

dos, sus nombres, que ninguno de ellos retorna nada y que uno tiene un puntero a *char* como atributo y otro, un entero.

Como nota final, y para concluir esta primera mirada sobre métodos y atributos, cabe mencionar que, al igual que las funciones, los métodos permiten la sobrecarga, la cual se rige bajo idénticas condiciones que las funciones.

Si nuestra clase fantasmita tuviera dos métodos define_color, uno aceptaría como parámetro el char, tal cual está definida actualmente, y otro aceptaría como parámetro un int, que va de 1 a 4, donde cada valor representa un color: 1: rojo; 2: verde; 3: amarillo y 4: azul. Así, nuestra clase quedaría definida con sus dos métodos de la siguiente manera:

```
1   #include <cstdlib>
2   #include <iostream>
3
4   using namespace std;
5
6   class fantasmita{
7
8   private:
9
10         char *color;
11         int estado; // 1 titilando 0 apagado
12
13  public:
14
15          void define_color(char *c);
16          void define_color(int c);
17          void define_estado(int e);
18  };
19
20  void fantasmita::define_color(char *c){
21
22        color=c;
23        cout <<"El color es "<<color<<endl;
24  }
25
26  void fantasmita::define_color(int c){
27        switch (c){
28                case 1:color = "rojo"; break;
29                case 2:color = "verde"; break;
30                case 3:color = "amarillo"; break;
31                case 4:color = "azul"; break;
32         }
33
34  cout <<"El color es "<<color<<endl;
35  }
36
37
38  void fantasmita::define_estado(int e){
39
40       estado = e;
41
42  }
43
44  int main(int argc, char *argv[])
45  {
46
47  fantasmita Rojo;
48  Rojo.define_color(1);
49
50  //Rojo.define_color("rojo");
51  // Rojo.estado=0; // no puedo acceder ya que el atributo es
    privado
52
53  return 1;
54  }
```

Script 9

Constructores

Una pregunta que siempre nos hacemos cuando estamos programando es qué valor tienen los objetos cuando se crean o qué valores deberían tener. Sabemos por experiencia que cuando creamos una variable propia y la instanciamos, la misma tendrá contenido basura hasta que le asignemos algún valor. Si instanciamos un objeto conocido, es posible que el mismo acepte valores para su inicialización y, de no ser pasados por parámetros, posiblemente tome valores por defecto.

Vamos a suponer que nos queremos comprar un perrito y, como lo vamos a pagar, pretendemos que el mismo posea algunas cualidades. ¿Qué valores nos gustaría decirle a la madre que configure en la creación de nuestro perrito? Supongamos que nos gustaría que fuera un labrador, negro, de ojos marrones, flaco y con un corte de pelo punk. Le especificamos, entonces, a la perrita nuestra sugerencia:

- Labrador.
- Negro.
- Ojos marrones.
- Flaco.
- Corte de pelo punk.

La perrita madre mira nuestras peticiones y configura su código de cría de perritos y escribe:

```
Nueva_cria Perrito (“Labrador”, “negro”, “marrones”, “flaco”, “punk”);
```

Con sólo observar la instanciación, podemos imaginarnos cuál es el atributo que modifica cada variable.

Seguramente, las primeras tres condiciones sean de por vida y no se puedan modificar. Sin embargo, las últimas dos irán cambiando a medida que vayamos criando al perrito.

Lo que aquí tenemos es un *constructor* del perrito. El constructor es un *método* que nos permite dar condiciones iniciales a nuestra clase, las cuales serán definidas en la instanciación del objeto. En general, el constructor suele especificar valores que no cambiarán a lo largo de la vida de nuestro objeto aunque también podrá alterar valores que luego podremos modificar.

Si tomamos como ejemplo nuestro fantasmita, vemos que tenemos dos atributos que nos gustaría definir en un estado inicial:

- color.
- estado.

El *color* es algo que elegimos al principio del juego y que vamos a mantener hasta que termine. Cada fantasmita tendrá su color y es único. El *estado* es algo que también tenemos que definir en un principio, pero a diferencia del *color,* no nos gustaría que se modifique en la instanciación. El fantasmita se iniciará en el estado de apagado y cuando Pac-Man coma la pastilla grande, podrá llamar a su método *come(),* quien verificará el atributo *pastilla_grande.* Entonces, recién ahí tendrá la posibilidad de llamar al método del fantasmita *define_estado(),* quien variará el estado del mismo por un tiempo determinado.

En otras palabras, queremos que nuestro fantasmita se inicie en un color rojo y en estado de apagado, pero deseamos que esto último sea por defecto y que no se modifique en los parámetros del constructor. Pronto veremos cómo hacer esto. Primero veamos cómo es la sintaxis de un constructor en C++:

```
class Mi_Clase{

      private:
```

```
            ...
            ...

      public:

         Mi_Clase(parametros);

             ...
             ...
             ...
};

        Mi_Clase::Mi_Clase(parametros){

          ...
          ...
            }
```

Script 10

El constructor es un método que lleva el mismo nombre que la clase y no tiene un valor de retorno.

A menos que la intención sea crear un objeto interno accesible sólo desde el interior de la clase, tenemos que cuidarnos en no declarar al constructor como privado, ya que de ese modo el mismo no podrá ser accedido en la instanciación.

Luego, desde el main, lo instanciamos así:

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

    Mi_Clase  Objeto1(parametros_iniciales);


    return 1;
}
```

Script 11

Ahora que ya sabemos cómo declarar un constructor, vamos a ver cómo hacemos las modificaciones propuestas a nuestra clase *fantasmita*.

```
1   #include <cstdlib>
2   #include <iostream>
3
4   using namespace std;
5
6   class fantasmita{
7
8   private:
9
10          char *color;
11          int estado; // 1 titilando 0 apagado
12
13
14  public:
15         fantasmita(char *c);     //constructor
16         void define_estado(int e);
17  };
18
19  fantasmita::fantasmita(char *c){          //declaracion del
```

```
    constructor
20
21              color=c;
22              estado = 0;
23
24              cout <<"El Fantasmita "<<color<<" quedo construido
    perfectamente"<<endl;
25  }
26
27  void fantasmita::define_estado(int e){
28
29        estado = e;
30
31  }
32
33
34  int main(int argc, char *argv[])
35  {
36
37       fantasmita Rojo("rojo");        //declaro el objeto  y paso
    un color como parametro
38
39  return 1;
40  }
```

Script 12

Tenemos, entonces, a nuestro *fantasmita* con su debido constructor. Como dijimos anteriormente, el mismo nos permite definir un color y establecer un estado por defecto. Notemos que de esta forma estamos obligados a pasar un color como parámetro en la instanciación de nuestro objeto *fantasmita*. En cambio, si llegáramos a dejar los paréntesis vacíos o con algún otro parámetro distinto al que acepta nuestro constructor, el compilador nos indicaría un error. Por lo tanto, ¿qué pasa si no quiero instanciar siempre el color de esa forma? Si deseáramos, por ejemplo, utilizar un valor del 1 al 4 para definir el color, además de la forma que ya usamos, podríamos sobrecargar el constructor. Al igual que cualquier método, el constructor se puede sobrecargar todas cuantas veces queramos, siempre y cuando no incurramos en una ambigüedad; incluso podemos crear un constructor que no acepte parámetros.

Definamos, ahora, otros dos constructores: uno que permita definir el color mediante un valor entero entre 1 y 4, y otro que no acepte parámetros y defina por defecto el color rojo:

```
1   #include <cstdlib>
2   #include <iostream>
3
4   using namespace std;
5
6   class fantasmita{
7
8   private:
9
10  char *color;
11  int estado; // 1 titilando 0 apagado
12
13
14  public:
15
16  fantasmita(char *c);  //constructor 1
17  fantasmita(int c);    //constructor 2
18  fantasmita();         //constructor 3
19
20  void define_estado(int e);
21
22  };
23
24  /* Constructor 1, acepta un char definiendo el color */
```

```
25
26  fantasmita::fantasmita(char *c){
27
28      color=c;
29      estado = 0;
30      cout <<"El Fantasmita "<<color<<" quedo construido
    perfectamente"<<endl;
31  }
32
33  /* Constructor 2, acepta un entero de 1 a 4 definiendo el color
    */
34
35  fantasmita::fantasmita(int c){
36
37  switch (c){
38         case 1:color = "rojo"; break;
39         case 2:color = "verde"; break;
40         case 3:color = "amarillo"; break;
41        case 4:color = "azul"; break;
42  }
43
44  estado = 0;
45  cout <<"El Fantasmita "<<color<<" quedo construido
    perfectamente"<<endl;
46
47  }
48
49  /* Constructor 3,no acepta parametros, define al color como rojo
    */
50
51  fantasmita::fantasmita(){
52
53  color="rojo";
54  estado = 0;
55  cout <<"El Fantasmita rojo quedo construido perfectamente"<<endl;
56  }
57
58  void fantasmita::define_estado(int e){
59  estado = e;
60  }
61
62  int main(int argc, char *argv[])
63  {
64
65  fantasmita Verde("verde");
66  fantasmita Amarillo(3);
67  fantasmita Rojo;
68
69  return 1;
70  }
```

Script 13

Aquí tenemos tres constructores distintos y en el main se puede observar cómo se crearon tres fantasmitas, cada uno con un tipo distinto de constructor.

Destructores

Siempre que hacemos algo en programación, tenemos la forma de deshacerlo.

Cuando creamos un objeto de manera dinámica (memoria dinámica), es decir, utilizando punteros, es muy deseable liberar esa memoria así como la de todas las variables alojadas de manera dinámica.

Para liberar la memoria de manera generalizada existe el *destructor*, un método similar al del constructor, pero al cual se le antepone el operador ~ y en la declaración del método, se indica qué memoria vamos a liberar.

Su sintaxis:

```
class Mi_Clase{

      private:

              ...
              ...


      public:

         ~Mi_Clase();

             ...
             ...
             ...

};

        Mi_Clase:: ~Mi_Clase(){

          ...
          ...

            }
```

Script 14

Los destructores se ejecutan automáticamente cuando el objeto se elimina. Si hubiésemos creado el objeto mediante un operador *new,* el mismo puede (y debe) ser eliminado de forma manual por el operador *delete*. El destructor no se puede sobrecargar, existe sólo uno.

Veamos un ejemplo completo de todo esto:

```
1   #include <cstdlib>
2   #include <iostream>
3
4   using namespace std;
5
6   class fantasmita{
7
8         /* Elementos privados */
9
10  private:
11
12          //atributos
13
14       char *color;
15       int * estado;
16
17         /* Elementospublicos */
18
19  public:
20
21  //Constructores y Destructor
22
23      fantasmita(char *c);  //constructor 1
24      fantasmita(int c);    //constructor 2
25      ~fantasmita();        //destructor
26
27  //metodos
28
29      void define_estado(int e);
30  };
31
32  /* Constructor 1, acepta un char definiendo el color */
```

```
33
34  fantasmita::fantasmita(char *c){
35
36          color=c;
37          estado = new int(0);
38          cout <<"El Fantasmita "<<color<<" quedo construido
    perfectamente"<<endl;
39  }
40
41  /* Constructor 2, acepta un entero de 1 a 4 definiendo el color
    */
42
43  fantasmita::fantasmita(int c){
44
45  switch (c){
46           case 1:color = "rojo"; break;
47           case 2:color = "verde"; break;
48          case 3:color = "amarillo"; break;
49          case 4:color = "azul"; break;
50  }
51
52  estado = new int(0);
53  cout <<"El Fantasmita "<<color<<" quedo construido
    perfectamente"<<endl;
54
55  }
56
57  // Declaracion del destructor
58
59  fantasmita::~fantasmita(){
60
61      delete estado;
62      color = NULL;
63      delete color;
64
65  cout <<"El fantasmita fue destruido perfectamente"<<endl;
66
67  }
68
69  // metodos
70
71  void fantasmita::define_estado(int e){
72
73  *estado = e;
74
75  }


76  int main(int argc, char *argv[])
77  {
78
79  fantasmita *Verde = new fantasmita("verde");
80  fantasmita *Amarillo = new fantasmita(3);
81
82  delete Amarillo;
83
84  return 1;}
```

Script 15

Ni los constructores ni los destructores son obligatorios. Si probaron los scripts habrán notado que hasta el 1.0.8, los constructores y los destructores no fueron usados y el objeto se declaraba a la perfección. Esto ocurre porque el compilador crea (en caso de que no hayan sido declarados) un constructor y un destructor por defecto que, obviamente, no hacen ninguna acción. El poder del lenguaje radica tanto en la posibilidad de hacer cosas, como en la de no hacerlas.

Como vemos, al ir conociendo nuevas herramientas, vamos ampliando nuestro conocimiento sobre las mismas y entendiendo que usarlas de manera correcta siempre es mejor que no utilizarlas.

Hasta aquí hemos visto de una manera muy simple lo que es la declaración de un objeto. Sin embargo no abordamos la parte más compleja, que es la interacción entre los mismos. Esta tarea es, sin dudas, una de las más difíciles, tanto para un novato, como para un gran experto. En la creación de los grandes software existen departamentos enteros que se dedican a esta etapa y hasta ahora no hay una forma óptima de hacerlo.

Por el momento nos vamos a detener aquí.

Ejercicio integrador

Como ejercicio integrador elegimos implementar un juego de magia. Para aprender cómo funciona, les sugerimos que tengan 21 cartas a mano.

- Se reparten las 21 cartas en tres columnas de siete cartas.
- Se le pide al espectador que elija una carta, sin decirnos de cuál se trata, pero indicándonos en qué columna está.
- Armamos cada una de las tres columnas en un macito de siete cartas y las volvemos a encimar para formar nuevamente el mazo de 21 cartas. *Pero*, cuando hacemos esto, dejamos en el medio a la columna elegida por el espectador.
- Desplegamos nuevamente el mazo en las tres columnas con siete filas y le pedimos nuevamente al espectador que elija.
- Juntamos otra vez, procurando dejar en el medio a la columna elegida.
- Hacemos lo mismo una última vez.
- Luego de repetir tres veces el procedimiento, contamos 11 cartas y *gualá,* la carta de nuestro espectador.

Si disfrazamos el truco con algún conejo y algo de ingenio a la hora de describir las cartas, podemos quedar muy bien.

Para programar dicho truco utilizamos números del 1 al 21 en lugar de cartas y definimos que un solo método se encargue de administrar todo.

El método *start* es el único método público y el que se encarga de administrar las manos.

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

using namespace std;

class magia{

      /* Elementos privados */

      private:
              // atributos

              int mesa[7][3];  // presentacion de la mesa
              int mazo[21];    // el macito de 21 cartas

             // metodos

              void  armar();       //arma  por  primera  vez  las  columnas
haciendo un random
              void mostrar();  //muestra la mesa
```

```
            int ingresar(int veces);          //es el ingreso de la
columna, tambien se encarga de mostrar la carta elegida
            void mezclar(int col);           //mezcla cada vez que se
elige una columna
            void repartir();                 //reparte el macito en la
mesa, no muestra
            void mostrar_carta_elegida(int g);       //muestra la carta
elegida

      /* Elementospublicos */

      public:

            magia();        //constructor, llama a "armar"
            void start();   //el unico metodo publico, es el que arranca
el juego y organiza las manos
            ~magia();       // destructor, vacio


};

        /**********************************/
        /* El constructor llama a armar, no lleva parametro*/
        /**********************************/

        magia::magia(){
           armar();
        }

        /**********************************/
        /*
        metodo: armar
        Arma la matriz de forma aleatoria
        primero pone los 21 valores ordenados y luego los cambia con
alguna coordenada de forma aleatoria
        */
        /**********************************/

       void magia::armar(){
            srand (time(NULL));
           int c=1;

           for (int i=0;i<7;i++){
               for(int j=0;j<3;j++){
                  mesa[i][j]=c++;
               }
           }

           for (int i=0;i<7;i++){
               for(int j=0;j<3;j++){

                  int filaux=rand()%7;
                  int colaux=rand()%3;
                  int aux=mesa[i][j];

                  mesa[i][j]=mesa[filaux][colaux];
                  mesa[filaux][colaux]=aux;
               }
           }

       }

       /**********************************/
        /*
        metodo: mostrar
        Arma un pequeno entorno grafico para mostrar la mesa
        */
        /**********************************/

       void magia::mostrar(){

           cout<<endl<<endl;
```

```
            for (int i=0;i<7;i++){
                cout<<"|";
                for(int j=0;j<3;j++){
                   cout<<setw(5)<<mesa[i][j]<<" | ";
                }
                cout<<endl;
            }
        cout<<endl<<setw(7)<<"col    1"<<setw(8)<<"col    2"<<setw(8)<<"col
3"<<endl;
        cout<<endl;

        }

         /**********************************/
         /*
         metodo: ingresar
         Pide la columna del espectador, recibe como parametro el nro de
mano
         Devuelve la columna elegida
         */
         /**********************************/


        int magia::ingresar(int veces){

             int col=0;
             bool vale=false;

             while (!vale){

                   cout<<endl<<"Ingrese el nro de columna en el cual esta
su numero escogido (1-3): ";
                   cin>>col;

                   if(col>0 && col<4)
                            vale=true;
             }

             if (2-veces)
                cout<<endl<<"quedan "<<2-veces<<" elecciones"<<endl;

             else {
                  mostrar_carta_elegida(col-1);
                  return -1;
             }
             return col;

        }

         /**********************************/
         /*
         metodo: mezclar
         se encarga de mezclar poniendo la columna elegida en el medio
         */
         /**********************************/


        void magia::mezclar(int col){
             int fil=0;
             int col_selec[3];

             switch(col){
                         case 1:
                              col_selec[0]=1;
                              col_selec[1]=0;
                              col_selec[2]=2;
                              break;

                         case 2:
                              col_selec[0]=0;
                              col_selec[1]=1;
```

```
                        col_selec[2]=2;
                        break;

                    case 3:
                        col_selec[0]=0;
                        col_selec[1]=2;
                        col_selec[2]=1;
                        break;
            }


            for(int j=0;j<3;j++){
                    for (int i=0;i<7;i++){
                        mazo[fil]=mesa[i][col_selec[j]];
                        fil++;
                    }
            }
        }


         /**********************************/
        /*
        metodo: repartir
        Acomoda el mazo en la mesa
        */
        /**********************************/

       void magia::repartir(){

            int fil=20;

            for(int i=0;i<7;i++){
                    for(int j=0;j<3;j++){

                        mesa[i][j]=mazo[fil];
                        fil--;

                    }
            }


       }

        /**********************************/
        /*
        metodo: mostrar_carta_elegida
        recibe como parametro la columna final, es llamado por mostrar
        */
        /**********************************/


      void magia::mostrar_carta_elegida(int g){



           cout<<endl<<"Ud eligio el: "<<mesa[3][g]<<endl<<endl;

           }


        /**********************************/
        /*
        metodo: start
        es el unico metodo publico, se encarga de administrar las manos y
llamar en orden a los metodos
        si c < 0 es porque ya termino el juego
        */
        /**********************************/

      void magia::start(){
```

```
            for (int juego=0;juego < 3;juego++){
                mostrar();
                int c=ingresar(juego);
                if (c>0){
                   mezclar(c);
                   repartir();
                }
            }
        }



        // Declaracion del destructor, vacio
        // no se uso memoria dinamica

        magia::~magia(){}


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

    magia M;
    M.start();


    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Script 16

El ejercicio propuesto sirve para ver de una manera integrada la incorporación de métodos, el control de acceso y la interacción entre los mismos. La simpleza del juego permitió que haya sido programado íntegramente en una sola clase, aunque la riqueza del paradigma radica en la integración de varios objetos simultáneamente. Por el momento, dejaremos esto para más adelante y nos conformaremos con aprender a dominar adecuadamente una única clase.

Bibliografía

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

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

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

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