UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño y programación de videojuegos

UNLVIRTUAL

Introducción a la programación

Unidad 7
Introducción a objetos

INTRODUCCIÓN

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.

Traigamos a colación nuevamente el ejemplo del juego de cartas donde dos jugadores manejan cada uno un mazo y deben sacar una carta, aquel con la carta más alta gana la mano y se lleva ambas cartas, la partida continúa hasta que uno de los jugadores se queda sin cartas. Cuando planteamos una solución para este juego dijimos que había varias cosas compartidas para cada jugador, es decir, si bien eran particular de cada jugador su comportamiento y uso tenía la misma lógica tanto para el Jugador número 1 como para el jugador número 2. Dicho esto, hicimos el siguiente esquema.

Jugador Número 1		**Jugador Número 2**
mazo		mazo
tomar carta		tomar Carta
	verificar mayor	
guardar carta		guardar carta

Dónde podíamos encontrar los siguientes conceptos comunes a ambos jugadores: Un **mazo**, una acción de **tomar una carta**, una acción de **guardar una carta**, entre otras cosas. Mientras que una acción de **verificar** cuál es la mayor sería común a ambos. Aprovechamos esta situación para entender el concepto de Funciones. La utilización de funciones nos permitió generalizar las acciones comunes y escribirlas una sola vez, aunque no nos resolvió el tema tener que llevar las variables para cada situación. Es decir, teníamos un mazo1, un mazo2, una carta1, una carta2. O sea, teníamos que llevar en una cuentita auxiliar todas las variables dependientes de cada jugador.

Imaginemos que podemos encerrar en una misma caja todos los conceptos pertenecientes a los jugadores:

Jugador Número 1
mazo
tomar carta
guardar carta

Jugador Número 2
mazo
tomar Carta
guardar carta

De esta manera el mazo del jugador 1 pertenece sólo al Jugador 1 y está definido sólo dentro de su caja. De la misma manera con las acciones. **Tomar carta** dentro de la caja del jugador 1 servirá sólo para los elementos del Jugador 1, no hace falta decirle de qué mazo tomar una carta, tampoco qué carta va a devolver.

Así, cuando queramos el mazo del jugador número 1 simplemente lo buscamos dentro de su caja. Cuándo queramos mezclarlo buscamos la función también dentro de la misma caja, etc.

OBJETOS EN C++

El concepto que acabamos de plantear es precisamente lo que se define como Objeto en la Programación Orientada a Objetos. Definir una caja general con variables y funciones que permita, a partir de ésta, crear cajas particulares: Jugador 1, Jugador 2, Mario, Luigi, Pac-man, Pikachu, etc.

En C++ un objeto se crea con la palabra reservada *Class*. De la siguiente manera:

```
class Jugador {
   …
```

```
}
```

Dónde, *class* es la palabra reservada que se usa para definir un objeto. Jugador es el nombre. Aquí se está definiendo el **tipo** de objeto, es decir, no estamos creando ningún Jugador en particular, sino que estamos definiendo cómo va a ser el tipo de dato Jugador (al igual que un int, un float, un string), estamos creando nuestro propio tipo de dato. Luego a partir de esta clase vamos a definir todos los jugadores que necesitemos. A continuación del nombre ponemos entre llaves todas las variables y funciones que pertenecen al objeto.

```
class Jugador {

    int mazo[];
    int carta;
    int sacarCarta();
    void guardarCarta();

}
```

Entonces, dentro de la clase Jugador tenemos dos variables, un arreglo de enteros llamado **mazo** y un entero que lleva la **carta**. También lleva dos funciones, **sacarCarta**, y **guardarCarta**. Ninguna de las dos funciones recibe parámetros ya que para este caso no hace falta debido a que ambas funciones trabajan con las variables definidas en el resto del objeto (mazo y carta). Las variables definidas en el objeto tienen su ámbito dentro de todos le objeto, por ende son accesibles para las funciones. Claro que si necesitamos algún valor externo sí debemos pasarlo por parámetro.

Ahora que ya conocemos la esencial vamos a aprovechar para hacer una aclaración. Las variables definidas dentro de una clase de llaman **atributos** de la clase, mientras que las funciones se denominan **métodos** de la clase. La implementación de los métodos se realiza de la misma manera que las funciones.

ATRIBUTOS

Los atributos son las variables dentro del objeto, su ámbito se extiende a todo el objeto. Por lo que no hace falta pasarlas por parámetro para que puedan ser modificadas en los métodos. Esto no quiere decir que los métodos no necesiten parámetros. Los necesitarán cada vez que tengan que utilizar un valor externo a la clase.

MÉTODOS

Los métodos de los objetos se comportan de la misma manera que las funciones, y como mencionamos, tienen acceso a los atributos de la clase. La declaración de un método es casi idéntica a la de una función, y puede hacerse de dos maneras:

La primera. A continuación de su definición (ejemplo método **sacarCarta**):

```
class Jugador {

    int carta;

    int sacarCarta(){
        return carta;
    };

};
```

La segunda. Por fuera de la clase. Se realiza de la misma manera, la única diferencia es debemos indicar a qué clase corresponde el método, esto lo

realizamos con los operadores :: (dos puntos, dos puntos) antepuestos al nombre de la clase:

```
class Jugador {

    int carta;
};
// aca finaliza la clase

    int  Jugador::sacarCarta(){
        return carta;
    };
```

Esta segunda manera facilita mucho la lectura, ya que nos permite ver claramente todos los métodos y atributos de la clase, y en caso de necesitar saber cómo se implementan, podemos ir a la implementación de manera separada.

Cómo ya mencionamos, los métodos tienen las mismas características de las funciones, corren las mismas reglas para los parámetros, se deben respetar los tipos, pueden hacerse por valor o por referencia, se pueden sobrecargar, etc.

UTILIZACIÓN Y ACCESO A LOS MÉTODOS Y ATRIBUTOS

Una vez declarada nuestra clase, es hora de instanciarla y acceder a sus métodos y atributos desde nuestro programa principal (main). Para ello instanciamos nuestra clase como cualquier otra variable:

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

    Jugador J1;

    return 0;
}
```

Aquí creamos una sola instancia (J1), pero podríamos haber creado todas las que quisiéramos, básicamente una para cada Jugador.

La sintaxis para instanciar es igual a cualquier otra variable, primero el tipo (en este caso el nombre de la clase: Jugador) y luego el nombre que le pongamos a esa variable, J1. Listo, ya tenemos una instancia de nuestra clase, J1 no es sólo una variable más, es oficialmente un **objeto** Jugador (así se denomina a las instancias de las clases).

Para acceder a los atributos y a los métodos usamos un operador punto (.) que nos dará acceso a la interface del objeto (es decir, su contenido, métodos y atributos). Desde ahí tenemos un comportamiento idéntico al de las variables y funciones.

```
    // recuperamos el atributo carta del objeto
J1
    int valor1 = J1.carta;

    // Le asignamos un valor al atributo carta
objeto J1
    J1.carta = 10;

    // Llamamos al metodo sacarCarta del objeto
J1
    // y recuperamos su valor en una variable
    int valor 2 = J1.sacarCarta();
```

En el último ejemplo, recuperamos el atributo carta del objeto J1, también lo modificamos y por último llamamos al método **sacarCarta**, el valor de retorno lo alojamos en otra variable.

MODIFICADORES DE ACCESO

Antes de continuar, haremos una pausa para aclarar algunos conceptos. Anteriormente, con la programación estructurada, nosotros hacíamos un programa mediante el desarrollo de un código fuente. Y el usuario es aquél que juega nuestro

juego. Ahora, con la programación Orientada a Objetos, nosotros hacemos primero las estructuras que luego utilizaremos en nuestro código. Básicamente estamos creando nuestros propios **tipos de datos** (claramente más complejos, que incluyen atributos y métodos). Dicho esto, vamos a resignificar nuestros usuarios. Los usuarios que utilizarán las clases que definamos serán otros programadores o quizás nosotros mismos a la hora de programar. Entonces, al igual que lo hacemos con un código usual, vamos a poner ciertas reglas sobre nuestras clases. Vamos a definir qué cosas se podrán acceder desde la instanciación qué cosas no.

¿Por qué querer permitir y negar acceso?

Pensemos la situación que veníamos teniendo, y damos libre acceso a todos los datos y métodos de nuestra clase. Podríamos si quisiéramos acceder directamente a la carta de cada Jugador y modificarla sin pasar por los controles necesarios de qué carta hay en el mazo o qué carta sacar, incluso podemos ponerle un valor fuera de rango. (Pensando siempre desde el punto de vista de la persona que va a usar la clase).

Para poder controlar el acceso o no a un método o atributo se usan **modificadores de acceso**. Los modificadores de acceso son simples palabras que le definen el permiso a los métodos y atributos. Los modificadores de acceso que existen en C++ (y la mayoría de los lenguajes) son:

- **public** (público)
- **private** (privado)
- **protected** (protegido)

Algo definido como **public**, podrá ser accedido sin problemas desde la instanciación del objeto. Algo definido como **private** no podrá ser accedido (será invisible, de hecho) desde la instanciación del objeto. Al modo **protected** lo veremos en la siguiente unidad.

```
class Jugador {

    private:

        int mazo[];
        int carta;

    public:

        int sacarCarta();
        void guardarCarta();

};
```

Aquí, ambos atributos son privados, mientras que ambos métodos son públicos. El ámbito del modificador se extiende desde el primer elemento (atributo o método) que se encuentra a continuación del mismo, hasta que aparezca otro modificador o bien la llave de cierre de la clase. En la instanciación:

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

    Jugador J1;

    // Esto es posible
    J1.sacarCarta();

    // Mientras que esto da error
    J1.carta = 10;

    return 0;
}
```

En la instanciación es perfectamente válido acceder al método **sacarCarta**, pero si queremos acceder de alguna manera al atributo carta, ya sea para leer su valor o modificarlo, el compilador dará error diciendo que ese atributo es privado.

En C++ si no colocamos ningún modificador, se toma por defecto que es privado. Esto quiere decir que lo que todo lo que escribimos antes de ver la existencia de los modificadores era siempre privado, por ende los códigos en la instanciación eran incorrectos, aunque sirvió para introducir el tema.

En resumen, esto:

```
class Jugador {

    private:

        int mazo[];
        int carta;

    public:

        int sacarCarta();
        void guardarCarta();

};
```

Es lo mismo que esto:

```
class Jugador {

        int mazo[];
        int carta;

    public:

        int sacarCarta();
        void guardarCarta();

};
```

Aunque personalmente soy de la idea de que siempre es mejor escribir todo para que el código quede bien claro y no haya chances de alteraciones ante algún copy paste o modificación indebida del código.

CONSTRUCTORES

Por lo general, cuando realizamos cualquier algoritmo, siempre está la necesidad de configurar las condiciones iniciales antes de meternos de lleno con el resto del algoritmo. En nuestro juego que venimos dando como ejemplo, sería la parte de inicializar los mazos para que tengan las cartas correspondientes cada uno, dar la primera configuración a la mesa, etc.

Cuando realizamos una clase, existe un método especial para realizar la configuración inicial y se denomina “Constructor”. El Constructor es un método más, con la diferencia que lleva el mismo nombre de la clase, debe ser siempre de acceso público y no tiene retorno. El constructor se ejecuta una sola vez, que es cuando se instancia el objeto. Luego no es posible volver a ejecutarlo, salvo que creemos el objeto nuevamente.

Podríamos declarar el constructor de la siguiente manera:

```
class Jugador {

    private:
        int mazo[];
        int carta;

    public:
        int sacarCarta();
        void guardarCarta();

        // Constructor
        // - Mismo nombre que la clase
        // - Publico
        // - Sin retorno
        Jugador();
};

        // En la declaracion
Jugador::Jugador(){

    int cantCartas = 48;
```

```
    for(int i=0;i<cantCartas;i++){
        mazo[i] = (i%12)+1;
    }
}
```

El constructor, cómo cualquier otro método, puede recibir parámetros y puede ser sobrecargado. Incluso, podemos definir distintas formas de instanciar un objeto dependiendo los parámetros del constructor.

En nuestro ejemplo, podríamos darle más generalidad a la clase y pasar por parámetro la cantidad de cartas que va a tener el mazo. Así el Jugador sirve para diferentes cantidades de cartas en el mazo:

```
class Jugador {

...
    public:
        Jugador(int cantCartas);
};

        // En la declaracion
Jugador::Jugador(int cantCartas){

    for(int i=0;i<cantCartas;i++){
        mazo[i] = (i%12)+1;
    }
}
```

Luego desde el main, al momento de la instanciación se llama automáticamente al constructor, si tenemos parámetros para pasar, podemos hacerlo poniendo los valores del constructor entre paréntesis, como veníamos haciendo con las funciones:

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

    Jugador J1(48);
    return 0;
}
```

Si el constructor no tiene parámetros, podemos omitir los paréntesis.

CLASES DENTRO DE CLASES

Lo más interesante de los objetos es poder reutilizarlos a nuestro antojo. Debemos recordar que lo que estamos haciendo aquí es nuestro propio tipo de dato, entonces si tenemos un objeto **rueda** y un objeto **auto** que tiene 4 objetos **rueda**. En nuestro objeto auto vamos a tener nuestros propios tipos de datos declarados dentro.

```
class rueda {
    public:
    int diametro;
    rueda(){};// Constructor nulo o vacio
    rueda(int d);
};
```

Luego nuestra clase Auto que tiene 4 ruedas.

```
class Auto {
public:
    rueda rd[4]; // 4 ruedas
    Auto(int d);
};
```

En la declaración del objeto tenemos un arreglo con ruedas. Para que esto funcione y

no nos de error nosotros tenemos que tener en la clase rueda un constructor vacío (también llamado nulo) ¿Por qué? Porque en la línea misma que declaramos del arreglo rueda, estamos haciendo una instanciación del objeto, y aunque todavía no tengamos los parámetros para ese objeto rueda (esos 4 objetos rueda) ya estamos reservando el espacio de memoria, así que estamos llamado a un constructor y el único constructor que coincide con esos parámetros es el constructor nulo.

Luego, cuando queramos darle valores a las ruedas, lo podemos hacer de la siguiente manera:

```
Auto::Auto(int t) {
    rd[0] = rueda(t);
    rd[1] = rueda(t);
    rd[2] = rueda(t);
    rd[3] = rueda(t);
}
```

A cada posición del arreglo le asignamos un nuevo objeto rueda recién instanciado, ahora sí llamado al constructor que queramos.

Luego en el main:

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

    Auto J(10);
    cout<<J.rd[2].diametro<<endl;

    return 0;
}
```

Esto funciona porque el arreglo asigna memoria de manera dinámica, pero si tenemos un elemento sólo y no usamos memoria dinámica (en la próxima unidad veremos esto). La solución que planteamos antes, no va a funcionar. Es decir, si tenemos en nuestra clase Auto una única rueda, no podemos declararla y asignarle un nuevo objeto luego (como lo hicimos con las posiciones de los arreglos). La forma de llamar a un constructor específico desde otro objeto es llamándolo desde la misma definición del constructor de clase contenida, separado por el operador: (dos puntos), así:

```
class Auto {
public:
    rueda rd; // Auto tiene una única rueda
    Auto(int d);
};
```

Luego en el constructor de Auto:

```
Auto::Auto(int t) : rd (t){

    cout<<"aqui el codigo del constructor de Auto"<<endl;

}
```

Notar que en la misma definición del constructor estamos llamando al constructor del objeto que queremos instanciar (¡¡ no a su clase !! al objeto ya declarado, es decir, no al constructor de rueda general, sino al del objeto instanciado en la clase **rd**)

```
Auto::Auto(int t)
```

Hasta aquí es el constructor nomal, pero le siguen los constructores de los objetos incluídos:

```
Auto::Auto(int t) : rd (t)
```

Aquí le estamos pasando el mismo valor que estamos recibiendo para el objeto auto, aunque podría tener cualquier otro, incluso algún valor por defecto:

```
Auto::Auto(int t) : rd (10)
```

¿Qué pasa si tenemos varios objetos?
Supongamos que en Auto tenemos una **rueda** y un **motor**, hacemos lo mismo pero los separamos por coma (,):

```
class Auto {
public:
    rueda rd; // Auto tiene una única rueda
    motor mt;
    Auto(int d);
};
```

```
Auto::Auto(int t) : rd (t), mt(40) {

    cout<<"aqui el codigo del constructor de Auto"<<endl;

}
```

Si ya conocemos los valores que le vamos a asignar a los objetos, es posible de usar otra forma de asignarle valores existentes a partir de C++ 11. Y es, asignándole los valores entre llaves en la misma instanciación:

```
class Auto {
public:
    rueda rd {10}; // Auto tiene una única rueda
    motor mt {50};
    Auto(int d);
};
```

ENCAPSULAMIENTO

El Encapsulamiento es una técnica relacionada intrínsecamente con la Programación Orientada a Objetos, se desprende directamente de la programación modular, y básicamente dice que el estado de un objeto sólo puede modificarse mediante las operaciones permitidas por ese objeto.

Es decir, como lo mencionamos antes, debemos recordar que los usuarios de las clases que hagamos podemos ser nosotros mismos u otros programadores. Supongamos que estamos encargados de realizar la clase que controla el lanzamiento de la palanca de las máquinas tragaperras de un casino. Y otro amigo programador utiliza la clase que nosotros hicimos para definirle los botones, los dibujos, etc. Si no somos cuidadosos, podríamos dejarle la libertad a nuestro amigo que utilice mal los datos, incluso sin quererlo.

El encapsulamiento tiene como objetivo definir métodos de entrada y salida de datos de la clase y que la única forma de comunicación de ese objeto sean mediante esos métodos. Como regla elemental se prohíbe el acceso a los atributos de manera directa (es decir que todos los atributos son privados). Si debemos permitir que un valor sea modificado directamente utilizamos métodos de *seteo* (set) y para obtener los valores, métodos *get*. Así, si queremos obtener el valor de una carta, podemos crear un método *getCarta* y que retorne su valor. De la misma manera podemos hacer un método que le defina un valor a la carta *setCarta*.

Si bien esto parece sólo una fachada para hacer lo mismo que podemos hacer accediendo directamente al dato, es una forma de protegernos contra nosotros mismos de descuidos.

Por ejemplo (volviendo al ejemplo del juego de cartas), aun si no permitimos que desde el exterior una persona pueda modificar el valor de la carta directamente, es útil crear un método privado que lo haga y utilizarlo nosotros mismos. El método *setCarta* puede tener ciertos controles, que la carta esté entre 1 y 12, que no esté repetida, etc. Así, si otro método falla e intenta modificar el valor de la carta a un valor incorrecto, evitamos que la falla se propague.

```
class Jugador {

    private:
        int carta;

        // modifica el valor de carta
        void setCarta(int car);


        // recupera el valor de carta
        int getCarta();

};

void Jugador::setCarta(int car){
    if(car >=1 && car <= 12){
```

```
        carta = car;
    }
}

int Jugador::getCarta(){
    return carta;
}
```