Programación Orientada a Objetos

Hasta el momento hemos estado trabajando bajo un paradigma de programación denominado programación estructurada y modular.

Esta forma de concebir la programación, y de producir sistemas a partir de ella, fue un gran avance en su momento ya que simplificó la forma de desarrollar software, frente a un mundo que demandaba más y más productos de este tipo. Sintéticamente podemos decir que lo que aporta la programación estructurada es la definición de las tres únicas estructuras de programación necesarias para resolver cualquier problema computacional: secuencia, decisión e iteración o repetición. La programación modular, por su parte, introduce el concepto de módulo, o subprograma (en C se implementan mediante funciones), que permite la descomposición del problema a resolver en un conjunto de módulos que interactúan.

Las funciones permiten abordar problemas complejos de manera sencilla, dividiéndolos en un conjunto de partes. En términos de programación podemos decir que invocar una función dentro de otra función de jerarquía superior, nos permite olvidarnos de los detalles y concentrarnos en la resolución del problema más general: basta llamar a la función pasándole los argumentos necesarios y esperar que ésta nos devuelva el resultado.

No obstante esto, las mejoras introducidas por el uso de la programación estructurada y modular resultaron insuficientes a la hora de producir elementos de software más complejos tales como animaciones, juegos interactivos, o programas muy grandes. En la programación estructurada todo el control queda a exclusiva responsabilidad del programador, y cada una de las líneas deben ser rigurosamente cuidadas, por lo que se desarrolló un nuevo enfoque o paradigma que se denominó Programación Orientada a objetos.

El nombre elegido tiene relación con lo que se buscaba: hacer más “natural” la programación, esto es, acercar la programación al lenguaje natural, que es el que utilizamos a diario.

La programación orientada a objetos parte de entender la realidad de la que formamos parte como el resultado de la interacción de un conjunto de objetos. Si aislamos una porción de esa realidad para representarla mediante un programa, lo que debemos hacer es definir cuáles son los objetos que interactúan en ella, y para cada uno de ellos, o para el conjunto de objetos de un mismo tipo, definir cuáles son sus características (sus propiedades o elementos constitutivos), y cuál es su comportamiento (que cosas hacen o necesitan hacer). Luego de eso basta con establecer cuáles son las interacciones que existen entre los objetos, cómo se relacionan entre sí o ante los estímulos que reciben de otros objetos, para construir un programa.

Para ir a lo concreto, supongamos que queremos hacer una animación simple: que un círculo rojo de 300 pixeles de radio se imprima en el centro de una pantalla de 1024*768 pixeles y se borre diez veces, en un determinado intervalo de tiempo.

En nuestra animación el único objeto que interviene es un objeto círculo. Sus propiedades son:

Color (para esta ocasión rojo)
Posición (un par de coordenadas x,y para el centro del cículo. Para este caso 512,384)
Tamaño (para este caso 300)

Y su comportamiento:

Mostrar (para que se imprima en la pantalla)
Ocultar (para que se borre de la pantalla)

Un posible código C++ para esa animación sería el siguiente:

```
int main(){
        Circulo c1(512, 384, 300, rojo, rojo);
        int i;
        for(i=0;i<10;i++){
                c1.Mostrar();
                delay(1000)
                c1.Ocultar();
                delay(1000);
        }
        return 0;
}
```

¿Extraño?. Por ahora sí.
Nosotros sabemos que si tipeamos este código en un cpp de CodeBlocks y lo compilamos va a mostrar más errores que líneas de código. Sin embargo no es un invento: podríamos conseguir la animación que se propuso si le agregáramos ciertas líneas de código.

Analicemos cada una de las líneas del código (se omiten aquellas cuyo significado ya se conoce):
Circulo c1(512, 384, 300, rojo, rojo); se declara un objeto Circulo y se establecen los valores de centro (512,384), tamaño (300) y los colores de relleno y borde (rojo).

c1.Mostrar(); se le pide al objeto Circulo que se muestre.

delay(1000); se detiene la ejecución del programa por 1000 milisegundos

c1.Ocultar(); se le pide al objeto Circulo que se oculte.

Y la accción de mostrar y ocultar se repite 10 veces por medio del for.

Supongamos que ahora queremos que se muestre el círculo alternativamente con los colores azul y amarillo. Debemos para ello agregar un nuevo comportamiento para el círculo, esto es, que tenga la capacidad de cambiar de color. Podría ser de la siguiente manera:

setColor(Color_de_borde, Color_de_relleno)

Nuestro programa sería entonces:

```
int main(){
	Circulo c1(512, 384, 300, rojo, rojo);
	int i;
	for(i=0;i<10;i++){
		if(i%2==0) c1.setColor(AZUL, AZUL);
		else c1.setColor(AMARILLO, AMARILLO);
		c1.Mostrar();
		delay(1000)
		c1.Ocultar();
		delay(1000);
	}
	return 0;
}
```

Si la variable que maneja el for es par (ó 0), se le pide al objeto que se pinte de azul; si es impar de amarillo. El resto del programa queda igual.

Algo similar podríamos hacer si quisieramos que el objeto cambia de tamaño (comportamiento setTamanio(int tamanio_pixeles)), o que cambie de posición (comportamiento setPosicionCentro(int valor_x, int valor_y)).

Supongamos que ahora queremos hacer otra animación usando círculos. Queremos que se muestren y se oculten 3 círculos de 200 píxeles distribuidos homogeneamente en la pantalla, con la siguiente distribución de colores (pueden elegir otros si no son de vuestro agrado): azul, amarillo, azul. El código sería:

```
int main(){
	Circulo c1, c2, c3; // se declaran los 3 círculos
	c1.setPosicionCentro(256,384);
	c2.setPosicionCentro(512, 384);
	c3.setPosicionCentro(768, 384);
	//se ubican los círculos homogeneamente en la pantalla
```

```
        c1.setTamanio(200);
        c2.setTamanio(200);
        c3.setTamanio(200);
        // se establece el tamaño de los círculos
        c1.setColor(AZUL,AZUL);
        c2.setColor(AMARILLO, AMARILLO);
        c3.setColor(AZUL,AZUL);
        // se establecen los colores de los círculos
        int i;
        for(i=0;i<10;i++){
                c1.Mostrar();
                c2.Mostrar();
                c3.Mostrar();
                delay(1000)
                c1.Ocultar();
                c2.Ocultar();
                c3.Ocultar();
                delay(1000);
        }
        return 0;
}
```

Analicemos ahora, a partir de lo que conocemos, el código:

Circulo c1, c2, c3;

Es la declaración de 3 variables de tipo Circulo. No es diferente a la declaración de variables que conocíamos hasta ahora; puede llamar la atención el tipo de dato (Circulo), pero lo mismo hacíamos cuando declarábamos variables de tipo struct (por ejemplo articulo reg1, reg2, reg3;). Lógicamente, para el caso de los structs sabemos que para declarar variables de ese tipo el struct tenía que estar definido en algún lado.

c1.setColor(rojo, rojo);
c1. Mostrar();

c1 es una variable de tipo Circulo; setColor(rojo, rojo) y Mostrar() son funciones. Y acá si que empiezan las "cosas extrañas": ¿puede una variable llamar a una función?. La respuesta es sí, en caso que la variable sea un objeto. En la programación orientada a objetos, **las variables no son simples recipientes de datos: también tienen un comportamiento, y por lo tanto podemos "pedirles que hagan cosas"**; si aquello que le pedimos es comprensible para ellas lo harán.

Antes de pasar a las definiciones formales analicemos lo siguiente: podríamos haber hecho las animaciones que se nos pidió, sin necesidad de saber cómo hace un círculo para

pintarse de un color o aparecer en la pantalla en una determinada posición. Es más, no nos interesa en el momento de hacer los programas cómo se consigue eso: nuestra atención se centró en conseguir construir la animación, sin interesarnos por los detalles, y en esto reside la potencia de la orientación a objetos, ya que nos provee de mecanismos de abstracción mucho más importantes que los de la pogramación estructurada y modular, donde sólo disponíamos de funciones para abstraernos de los detalles.

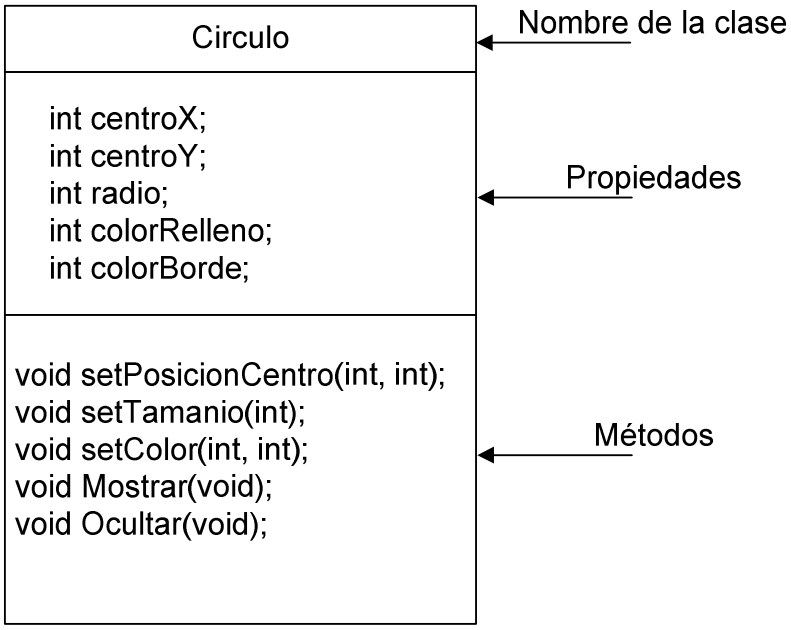
Ahora sí las definiciones. Dijimos que c1, c2 y c3 son variables objetos de un mismo tipo o una misma clase. Definamos entonces objetos y clases:

Objeto: entidad del mundo real o virtual sobre el cual se desea registrar información, o que forma parte de la porción de la realidad que se quiere representar mediante un programa de computadora. En términos de programación son variables de una determinada clase. También se define a un objeto como una instancia de una clase.

Clase: plantilla o molde adonde se definen las características (propiedades) y el comportamiento (métodos) de todos los individuos pertenecientes a esa clase. En términos de programación las propiedades son las variables de la clase, y los métodos las funciones de la clase; una clase es la definición de un tipo de dato .

Podemos hacer una analogía entre las clases y los objetos a partir de los structs. Por ejemplo, para declarar variables de tipo struct articulo, primero debemos hacer el molde, o la definición del struct articulo. Para declarar objetos Circulo, primero debemos construir el molde, es decir, la clase Circulo.

En la notación UML podemos representar la clase Circulo de la siguiente manera:



Como ya se dijo, si nos hubieran dado la clase Circulo (por ejemplo en un .h), y su especificación (lo que se nos muestra en el diagrama anterior: las propiedades y los métodos), podríamos haber hecho las animaciones del ejemplo sin necesidad de conocer el código asociado.

Todos los objetos de la clase Circulo tendrán las propiedades definidas en la clase, y podrán llamar a los métodos definidos dentro de la clase.

Construcción de clases

Veamos ahora en un ejemplo más simple que el de la clase Circulo como construir una clase.

Supongamos que deseamos representar mediante una clase objetos articulo, tal como el que definimos para trabajar con structs. Sus propiedades serán

```
char codArticulo[5];
char descripcion[30];
float pu;
int stock;
```

Y como métodos, por ahora vamos a utilizar uno que permita asignarle valores a las propiedades, y otro para mostrar por pantalla el valor de éstas:

```
void Cargar(void);
void Mostrar(void);
```

La clase Articulo sería de la siguiente manera:

```
class Articulo{
        private:
                char codArticulo[5];
                char descripcion[30];
                float pu;
                int stock;
        public:
                void Cargar(void);
                void Mostrar(void);
        };
```

Para definir una clase se utiliza la palabra clave class y luego el nombre que le querramos asignar. Luego entre las llaves se definirán las propiedades y los métodos. La llave de cierre, al igual que en los structs lleva un punto y coma.

Dentro de la clase las propiedades y métodos se ubican en lugares distintos: las propiedades luego de private:, y los métodos luego de public:; private y public son los especificadores de acceso, y establecen que será o no accesible fuera de la clase:

private: (privado) establece que los miembros de la clase definidos son privados, lo que significa que no serán accesibles fuera de la clase.

public: (público) establece que los miembros de la clase definidos son públicos, lo que significa que serán accesibles fuera de la clase.

La diferencia que se establece entre propiedades en la parte privada, y métodos en la parte pública no tiene que ver con requerimientos de programación. Ninguna clase va a dar errores si se ponen las propiedades en la parte pública, o si se colocan métodos en la parte privada. Pero aquí es donde aparece uno de los principios de la Programación Orientada a Objetos, que se denomina **encapsulamiento**.

Dijimos que un objeto tiene propiedades (variables de la clase que almacenan datos), y métodos (funciones de la clase que determinan su comportamiento). También dijimos que los objetos tienen la capacidad de hacer lo que se le pida, siempre y cuando esto esté definido dentro de la clase. El **encapsulamiento es la capacidad de un objeto de mantener ocultos y preservados sus datos**, a fin de garantizar que no puedan ser modificados accidentalmente, o que se le asignen valores incorrectos.

Por ejemplo, en el programa de la animación de los círculos, se hizo lo siguiente para modificar la posición:
c1.setPosicionCentro(256, 384);

si las propiedades centroX y centroY hubieran sido públicas podría haberse escrito de esta otra manera:

c1.centroX=256;
c1.centroY=384;

y lograrse el mismo resultado. Sin embargo esta última opción rompe con el encapsulamiento, y no le da la posibilidad al objeto que se reponga de una posible mala asignación, como la siguiente:

c1.centroX=1500;
c1.centroY=1000;

ya que estas coordenadas se encontrarían fuera de la pantalla. Si esta asignación se hubiera hecho con el método setPosicion()
c1.setPosicion(1500, 1000);

el método podría tener un remedio para evitar el problema (por ejemplo dejar los valores de centro sin modificar, ubicar el círculo en una determinada posición, generar un mensaje de error, etc.).

Para evitar esto (las asignaciones incorrectas), siempre se pone las **propiedades como privadas**, y sólo se permite su modificación mediante métodos de la propia clase. Esta es una norma central de la progamación orientada a objetos que respetaremos sin excepción en nuestro curso.

Volviendo al diseño de la clase Articulo, vemos que dentro de ella en la parte pública hemos puesto los prototipos de las funciones o métodos Cargar() y Mostrar(). Podríamos haber puesto allí la definición completa de los métodos, pero por una cuestión de estilo y para darle mayor facilidad a la lectura del código (no poner todo junto) haremos la definición de las funciones fuera de la clase de la siguiente manera:

```
void Articulo::Cargar(){
        cout<<"INGRESE EL CODIGO: ";
        cin>>codArt;
        cout<<"INGRESE LA DESCRIPCION: ";
        cin>>descripcion;
        cout<<"INGRESE EL PRECIO UNITARIO: ";
        cin>>pu;
        cout<<"INGRESE EL STOCK";
        cin>>stock;
        }

void Articulo::Mostrar(){
        cout<<" CODIGO: ";
        cout<<codArt<<endl;
        cout<<" DESCRIPCION: ";
        cout<<descripcion<<endl;
        cout<<" PRECIO UNITARIO: ";
        cout<<pu<<endl;
        cout<<" STOCK";
        cout<<stock<<endl;
        }
```

En la cabecera de la definición de los métodos, se usó la siguiente sintaxis:

```
void Articulo::Mostrar()
```

donde:
void: valor que devuelve la función

Articulo: clase a la que pertenece la función
:: operador de alcance de resolución (scope). Indica que la función que se escribe a continuación del operador pertenece a la clase que se detalla antes del operador.
Cargar(): nombre de la función. Si recibiera parámetros, deberían haberse puesto entre los paréntesis de la función.

Si analizamos el código de las funciones observamos que éstas usan directamente las variables de la clase sin declararlas. Esto sucede porque a diferencia de las funciones que conocíamos hasta el momento (que vamos a llamar a partir de ahora funciones globales), las funciones son miembros de la clase, y como tal conocen las variables definidas dentro de su misma clase. Por esta razón (su pertenencia a una clase) los métodos sólo pueden ser llamados por los objetos de esa clase. Veamos un ejemplo:

Ejemplo 1:
Hacer un programa para cargar y mostrar un objeto artículo.

```
int main(){
        Articulo obj;
        obj.Cargar();
        obj.Mostrar();
        return 0;
        }
```

Como en el ejemplo de los círculos vemos que es el objeto el que se encarga de realizar las acciones. No cargamos o mostramos individualmente cada una de las propiedades como lo haríamos con un struct: le pedimos al objeto que lo haga. El código que permite trabajar con cada propiedad se encuentra dentro de la clase, en los métodos, y en ellos se podrían establecer las restricciones o validaciones que se consideren necesarias.

Como las funciones pertenecen a las clases, y sólo pueden ser llamadas por objetos de esa clase, podríamos tener otras clases con los mismos nombres de funciones sin que esto genere inconvenientes. Por ejemplo:

```
…….
Articulo arti;
Cliente cli;
Venta ven;
arti.Cargar();
cli.Cargar();
ven.Cargar();
……..
```

Cada objeto llama a un método de nombre Cargar(), pero como son objetos de distinta clase, en realidad están llamando a funciones diferentes (las definidas dentro de cada clase).

Como se dijo, las propiedades no son accesibles fuera de la clase, ya que están declaradas en la parte privada de la clase. Ahora, ¿cómo se hace para acceder a una propiedad en particular fuera de la clase?. La respuesta es mediante métodos específicos de la propia clase. Veamos un ejemplo:

Ejemplo 2:
Dados dos objetos artículo, asignarle valores y luego informar si los códigos son iguales.

```
int main(){
        Articulo obj1, obj2;
        obj1.Cargar();
        obj2.Cargar();
        if(strcmp(obj1.getCodArt(), obj2.getCodArt())==0)
        cout<<"LOS CODIGOS SON IGUALES";
        return 0;
        }
```

Ambos objetos llaman al método getCodArt(), que devuelve el valor del código de cada artículo. Como los códigos son cadenas, debe usarse strcmp() para compararlos. El método utilizado tiene el siguiente código.

```
char * Articulo::getCodArt(){
        return codArticulo;
        }
```

Del mismo modo podrían hacerse métodos que devuelvan el valor de cada una de las propiedades, para que puedan ser conocidas fuera de la clase.

```
char * Articulo::getDescripcion(){
        return descripcion;
        }

float Articulo::getPu(){
        return pu;
        }

int Articulo::getStock(){
```

```
        return stock;
        }
```

Los métodos podrían haber sido definidos con cualquier nombre válido. Se optó por usar get unido al nombre de la propiedad, porque es la sintaxis que utilizan la mayoría de los lenguajes de programación para estos métodos.

La otra dificultad que se nos presenta debido a la privacidad de los datos es que tampoco podemos modificarlos directamente. También en este caso será necesario crear los métodos para cada propiedad:

```
void Articulo::setCodArt(char *cod){
        strcpy(codArticulo, cod);
        }

void Articulo::setDescripcion(char *desc){
        strcpy(descripcion, desc);
        }
void Articulo::setPu(float p){
        pu=p;
        }

void Articulo::setStock(int s){
        stock=s;
        }
```

Vectores y punteros de objetos

Una clase permite declarar variables (objetos) de ese tipo. Un clase es un tipo de dato definido por el usuario (se lo define como TDA: tipo de dato abstracto), y al igual que para el resto de los tipos de datos se pueden declarar variables simples, vectores, matrices y punteros. También pueden ser enviados como parámetros entre funciones, o ser devuelto por una función, de acuerdo a las reglas que ya conocemos.

Ejemplo 3:
Dada una lista de 10 artículos, cargarla en un vector y luego mostrar el contenido.

```
…….
void cargarVectorArticulos(Articulo *);
void mostrarVectorArticulos(Articulo *);

int main(){
        Articulo vart[10];
        cargarVectorArticulos (vart);
        mostrarVectorArticulos (vart);
```

```
	return 0;
	}

void cargarVectorArticulos(Articulo *v){
	int i;
	for(i=0;i<10;i++)
		v[i].Cargar();
	}

void mostrarVectorArticulos(Articulo *v){
	int i;
	for(i=0;i<10;i++)
		v[i].Mostrar();
	}
```

En el caso del uso de vectores, el vector no será el objeto, sino cada una de sus componentes. El vector oficia de contenedor de objetos.