Introducción

Hasta el momento hemos representado elementos abstractos tales como listas de números, caracteres, etc y elementos concretos del mundo real, por ejemplo: alumnos, vehículos, multas, artículos, facturas de venta, entre otros. Es muy común que en la programación sea necesario representar este tipo de objetos complejos.

En la vida real, estos elementos tienen características que lo identifican y comportamientos que realizan. Por ejemplo, un automóvil tiene una marca, año de fabricación, color, cantidad de puertas, etc. Si lo llevamos a un programa, estas características van a depender de qué tipo de sistemas tengamos que representar. Podemos decir que dependen de un contexto. Lo mismo ocurre con el comportamiento, las acciones que puede hacer un automóvil en nuestro programa va a depender del tipo de programa. Desde darnos la posibilidad de cargar la información de nuestro vehículo hasta guardarlo en un archivo si es necesario.

Este contexto es muy importante para poder modelar correctamente el objeto en nuestro programa. En un sistema que gestione alquileres de autos, los atributos del automóvil serán los necesarios para gestionar el sistema: patente, marca, color, modelo, año de patentamiento, kilómetros recorridos, etc y las acciones que podrá hacer un automóvil probablemente tendrán relación a poder gestionar esta información en memoria y almacenarlos en archivos o bases de datos. En cambio, si estuvieses diseñando un juego de carreras nuestro concepto de automóvil sería distinto porque el contexto es otro. Tal vez las características serán marca, modelo y color. Pero es posible que tengan otros elementos como vidas o armamento además de las distintas imágenes que serán necesarias para animar el objeto y dar la sensación de movimiento. El comportamiento del automóvil en un juego será distinto, tendrá que ser capaz de mostrarse y ocultarse, también probablemente necesitará acelerar y frenar, entre otras cosas.

Este tipo de representación la lograremos mediante el uso de clases.

Programación estructurada vs Programación orientada a objetos

La programación estructurada nos ayuda a mejorar la claridad y facilidad para programar. Haciendo uso de estructuras como la secuencia, decisión y repetición es posible elaborar un

algoritmo eficiente que resuelva un problema de una manera sencilla. Si a esto le sumamos la capacidad de dividir las tareas en pequeños módulos o funciones, entonces logramos un producto mucho más fácil de mantener, entender y reutilizar. Sin embargo, si tenemos que representar algo mucho más complejo, la rigurosa tarea de codificar cada elemento de nuestro programa en este paradigma será muy difícil. Ahí es donde la programación orientada a objetos nos será de utilidad.

La POO a diferencia de la anteriormente mencionada es un paradigma en el que se enfoca en modelar los diferentes objetos que intervienen en el problema, describiendo sus relaciones entre ellos, sus atributos y comportamientos. Una vez analizados, modelados y codificados estos objetos, la interacción entre ellos será la que nos permitirá resolver nuestros algoritmos.

Pensemos el siguiente ejemplo, supongamos que queremos cargar la información de un examen y mostrarla por pantalla. Necesitaremos pensar los elementos que representan a un examen en nuestro sistema, por ejemplo: Apellido y nombre del alumno, fecha del examen, nombre de la materia y calificación. Si intentamos representarlo con variables simples tendríamos algo similar a lo siguiente:

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

int main(){
  int dia, mes, anio, nota;
  char apellido[50], nombre[50], materia[50];
  cout << "Apellido: ";
  cin >> apellido;
  cout << "Nombre: ";
  cin >> nombre;
  cout << "Materia: ";
  cin >> materia;
  cout << "Fecha (DD MM AAAA): ";
  cin >> dia >> mes >> anio;
  cout << "Calificación: ";
  cin >> nota;

  cout << endl << apellido << ", " << nombre << " - " << dia << "/" <<
  mes << "/" << anio << " - Calificación: " << nota << endl;
  return 0;
}
```

Código 1 · Posible representación de la carga y listado de un examen.

Puede resultar no tan complejo, pero recordemos que estamos representando sólo un examen. Si nuestro objetivo es tener los exámenes de un curso. Tendríamos que enredarnos entre distintos tipos de arrays para poder representar la información.

La manera en que abordaríamos este problema con la POO es primero pensar qué es un examen para nuestro sistema. ¿De qué características está compuesto? y ¿qué acciones se podrán realizar con un examen en nuestro sistema? Una vez que tengamos esto claro, podremos comenzar diseñando nuestra *clase*.

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

class Examen{
  private:
    int dia, mes, anio, nota;
    char apellido[50], nombre[50], materia[50];
  public:
    void cargar(){
      cout << "Apellido: ";
      cin >> apellido;
      cout << "Nombre: ";
      cin >> nombre;
      cout << "Materia: ";
      cin >> materia;
      cout << "Fecha (DD MM AAAA): ";
      cin >> dia >> mes >> anio;
      cout << "Calificación: ";
      cin >> nota;
    }
    void mostrar(){
      cout << endl << apellido << ", " << nombre << " - " << dia <<
      "/" << mes << "/" << anio << " - Calificación: " << nota;
    }
};

int main(){
  Examen reg;
  reg.cargar();
  reg.mostrar();
  return 0;
}
```

Código 2 · Solución del problema del examen utilizando una clase Examen.

Pareciera más trabajo, más código, más palabras reservadas por aprender y entender. Pero las soluciones que traerá a la larga serán mayores que las complicaciones que puede generar ahora.

Declaración de una clase

Comencemos por analizar cómo declarar una clase.

```
class NombreClase{
  private:
   /// Declaración de los atributos (variables y/o constantes) y métodos
   /// de nuestro objeto que no pueden ser accedidos desde fuera de la
   /// clase.
  public:
   /// Declaración de los atributos (variables y/o constantes) y métodos
   /// de nuestro objeto que pueden ser accedidos desde fuera de la
   /// clase.
};
```

Código 3 · Sintaxis abreviada de cómo se puede declarar una clase.

Declarar una clase es indicar cómo se llama, cuáles son los atributos que caracterizarán a los objetos creados a partir de ella y cuáles serán los métodos que realizará.
Podrán notar que los atributos de un objeto son variables y/o constantes (de tipos de datos simples o de otros objetos) y los métodos son funciones que nuestros objetos podrán ejecutar. Estos últimos podemos pensarlos como funciones de la clase y a las que actualmente conocemos como funciones las llamaremos funciones globales.
Es importante acostumbrarse que una buena práctica en la POO es que los atributos de una clase sean privados y que para poder accederlos lo hagamos a través de métodos que son públicos. Esto se verá con más detalle cuando se aborde el Encapsulamiento.

¿Qué diferencia hay entre una clase y un objeto?

La diferencia entre clase y objeto es sustancial y es importante conocerla para no cometer errores en nuestro código. Mientras la clase define cómo será la estructura de atributos y métodos, el objeto es una variable en memoria al cual se le puede asignar valores y llamar a sus métodos.
Podemos decir que con una clase definimos un tipo de dato nuevo. Del cual podremos crear objetos. Y un objeto es una variable (o instancia) de una clase en particular.
Si tenemos las siguientes líneas de código:

```
int entero;
Examen exam;
```

Entonces *Examen* e *int* son tipos de datos mientras que entero y *exam* son *variables*.

¿Se puede dividir el código de una clase en H y CPP?

Sí, es muy común. En este caso desarrollaremos la declaración de la clase dentro del archivo h y la definición de los métodos en el archivo cpp.

```
class Examen{
  private:
    int dia, mes, anio, nota;
    char apellido[50], nombre[50], materia[50];
  public:
    void cargar();
    void mostrar();
};
```

Código 4 · Archivo examen.h

```
#include <iostream>
using namespace std;
#include "examen.h"

void Examen::cargar(){
  cout << "Apellido: ";
  cin >> apellido;
  cout << "Nombre: ";
  cin >> nombre;
  cout << "Materia: ";
  cin >> materia;
  cout << "Fecha (DD MM AAAA): ";
  cin >> dia >> mes >> anio;
  cout << "Calificación: ";
  cin >> nota;
}
void Examen::mostrar(){
  cout << endl << apellido << ", " << nombre << " - " << dia <<
  "/" << mes << "/" << anio << " - Calificación: " << nota;
}
```

Código 5 · Archivo examen.cpp

Podemos notar, a diferencia del *Código 3*, que si separamos la clase en declaración y definición. Al definir los métodos tenemos que indicar que los mismos son funciones de clase. Más precisamente de la clase Examen. Para ello se utiliza el nombre de la clase y el operador de ámbito `::` ya que de lo contrario estaríamos definiendo una función global y no un método de clase. En el *Código 3*, no hubo necesidad de realizar esta distinción porque definimos el método en el mismo momento que se declaró la clase. Es decir, dentro de las llaves de la misma.

Por último, main.cpp nos quedaría de la siguiente manera:

```
#include "examen.h"

int main(){
  Examen reg;
  reg.cargar();
  reg.mostrar();
  return 0;
}
```

Código 6 · Archivo main.cpp

Aclaración: Los métodos cargar y mostrar de la clase Examen son utilizados en este ejemplo para visibilizar la capacidad que tiene un objeto de poder llamar métodos (o funciones de clase). Luego, en el futuro, no será una buena práctica que una clase como Examen tenga tales métodos. Ya que si un examen debe cargarse o mostrarse a través de la consola de una manera determinada será una clase la que tenga la facultad de hacer tales acciones. Por ejemplo, una clase llamada ExamenManager podría ser quien pueda cargar un examen o mostrarlo por pantalla.