En las páginas anteriores hemos visto cómo diseñar una clase y los conceptos fundamentales de porque utilizarlas. Y una de las tantas cosas nuevas que surgieron es el concepto de ámbito.

Anteriormente, el ámbito podía ser local a una función o una estructura (un if, for, while, etc) o bien un ámbito global.

En la POO, cuando declaremos una clase definiremos el ámbito de sus atributos y métodos miembro. Los ámbitos pueden ser private, public, protected o friend. En este apunte se detallarán las características de los especificadores de acceso private y public.

En el Código 2 del apunte, se definieron los atributos como privados y los métodos cargar y mostrar como públicos. Si quisieramos ejecutar el siguiente código de main utilizando esa clase obtendríamos un error:

```
int main(){
  Examen reg;
  cin >> reg.apellido;
  reg.nota = 10;
  return 0;
}
```

**Código 7** · Programa main con errores de acceso a miembros privados de la clase Examen.

```
error: 'char Examen::apellido [50]' is private within this context
error: 'int Examen::nota' is private within this context
```

**Salida 1** · Mensajes de error generados por el compilador.

Podemos deducir que los miembros declarados como privado no pueden ser accedidos desde fuera de la clase. En cambio, aquellos declarados como públicos sí. Esto no es un error ni un inconveniente sino una característica de la programación orientada a objetos llamada **encapsulamiento**.

## Encapsulamiento

Esta característica de la POO sólo los elementos definidos como **public** pueden ser accedidos desde fuera de la clase. Mientras que los definidos como **private** son inaccesibles y, por lo tanto, se encuentran encapsulados si se quieren acceder desde fuera de la clase. El concepto general es que al definir una clase debemos idear cuál es la interfaz con la cual se comunicará con el resto del sistema (otros objetos y funciones del sistema). Este medio de comunicación es proporcionado por los elementos que hayan sido declarados públicos. En el ejemplo de la clase Examen, la manera de interactuar con ella desde afuera (función main) es mediante los métodos cargar y mostrar (que anteriormente ya mencionamos que es mejorable con la ayuda de otra clase). Quiere decir que no tenemos manera de acceder a cada uno de los elementos de la clase por separado, sólo permite cargarse completamente por ingreso directo del usuario y mostrar todos sus métodos por pantalla.

Si quisiéramos acceder a cada uno de los atributos de esta clase tanto para leerlos como para cambiarlos lo correcto sería hacerlo a través de una serie de métodos que lo permitan. Este tipo de métodos son tan comunes que suelen conocerse como *getters* y *setters*.

## Setters y getters

Este tipo de métodos no tienen nada de especial. De hecho, podrían ser llamados de cualquier manera. Incluso *castellanizarlos* y llamarlos obtenedores y establecedores. Pero se acostumbra llamar `setNombreAtributo` al método que permite asignarle un valor al atributo en cuestión y `getNombreAtributo` al método que permite obtener dicho valor.

Podría ocurrir que un atributo en particular no tenga un setter y sí un getter o viceversa. Todo dependerá del contexto y lo que necesitemos que nuestra clase permita hacer con los datos. Veamos un ejemplo con la clase Examen:

```
#ifndef EXAMEN_H
#define EXAMEN_H
class Examen{
  private:
      int dia, mes, anio, nota;
      char apellido[50], nombre[50], materia[50];
  public:
      const char * getApellido();
      void setApellido(const char *);
      int getNota();
      void setNota(const int);
      // Declaraciones de los demás métodos setters y getters.
```

```
      void cargar();
      void mostrar();
};
#endif // EXAMEN_H
```

**Código 8** · Declaración de clase Examen con declaración de setters y getters.

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

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

void Examen::setApellido(const char *nuevoApellido){
  strcpy(apellido, nuevoApellido);
}
const char * Examen::getApellido(){
  return apellido;
}
void Examen::setNota(const int nuevaNota){
  if(nuevaNota >= 1 && nuevaNota <= 10){
    nota = nuevaNota;
  }
  else{
    nota = 0;
  }
}
int Examen::getNota(){
  return nota;
}
```

**Código 9** · Definición de métodos de la clase Examen. Se pueden observar el detalle de la acción de los setters y getters.

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

int main(){
  Examen reg;
  reg.cargar();
  cout << reg.getApellido() << " - " << reg.getNota();
  return 0;
}
```

**Código 10** · Función main haciendo uso de los getters para mostrar por pantalla el apellido y nota cargadas previamente por el método cargar.

Como podemos observar en los ejemplos, el encapsulamiento nos provee una manera de trabajar con las clases de forma tal que no se puedan acceder a los datos si la misma no lo autoriza. O como se puede notar en el método *setNota,* que es la única manera de cargar una calificación a un objeto de la clase Examen, no se puede asignar una valor que no se considere correcto (número entero entre 1 y 10); en caso contrario se establece una nota cero que podría significar que no existe una nota válida cargada.

Sin embargo, qué ocurriría si ejecutamos el siguiente código en main:

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

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

```
O"E, ??? - 1161838592/32756/1160649104 - Calificación: 32756
```

**Salida 2** · Valores de los atributos del objeto reg de la clase Examen.

La salida por pantalla generada es de esperarse. Es un comportamiento normal cuando declaramos una variable y no inicializamos. Pero en este caso, no se trata de una variable simple que podamos igualar a cero. Sino de una variable compleja u objeto que puede contener muchos elementos. Una manera de solucionar este tipo de problemas es con un método especial que tienen las clases llamados constructores.

## Constructor y destructor

Un constructor es un método de clase que tiene la característica de que se llama automáticamente cuando un objeto es instanciado. Es decir, cuando se declara la variable o se pide memoria para un objeto.

Declarar y definir un constructor es muy simple. Es necesario crear un método que se llame igual que la clase y que no devuelva nada (ni siquiera void). El código del constructor puede ser cualquiera. Se suele utilizar para inicializar el objeto con valores por defecto (o recibidos por parámetro) aunque también puede ser útil para pedir memoria dinámica.

Por otro lado, un destructor es un método de clase que se llama cuando el objeto está a punto de ser eliminado. Es decir, el alcance del objeto finaliza o se libera memoria dinámica solicitada previamente. Para declarar un destructor es necesario el caracter de virgulilla (~) y el nombre de la clase.

```
#ifndef EXAMEN_H
#define EXAMEN_H
class Examen{
  private:
      int dia, mes, anio, nota;
      char apellido[50], nombre[50], materia[50];
  public:
      Examen();
      Examen(int, int, int);
      int getNota();
      void setNota(const int);
      void mostrar();
      ~Examen();
};
#endif // EXAMEN_H
```

**Código 11** · Declaración de clase Examen con declaración de dos constructores y un destructor.

En el código 11 se puede visualizar la declaración de la clase Examen con dos constructores (no existe ningún problema en tener métodos con mismo nombre mientras se diferencien por los parámetros que reciben → ver Sobrecarga). También se puede observar la declaración del destructor.

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

Examen::Examen(){
  dia=mes=anio=nota=0;
  strcpy(materia, "");
  strcpy(apellido, "");
  strcpy(nombre, "");
}
Examen::Examen(int d, int m, int a){
  dia=d;
  mes=m;
  anio=a;
  nota = 0;
  strcpy(materia, "");
  strcpy(apellido, "");
  strcpy(nombre, "");
}
Examen::~Examen(){
  cout << "Llamado al destructor de examen" << endl;
}
void Examen::mostrar(){
  cout << endl << apellido << ", " << nombre << " - " << dia <<
  "/" << mes << "/" << anio << " - Calificación: " << nota;
}
```

**Código 12** · Definición de métodos de la clase Examen, incluyendo dos constructores y un destructor.

En el código 12 se puede visualizar la definición de la clase Examen con dos constructores. Se diferencian entre ellos ya que el constructor que recibe parámetros establece la fecha del examen a partir de los parámetros. En cambio, el constructor sin parámetros sólo se encarga de poner todos los valores enteros en cero.

El destructor, se encarga de mostrar por pantalla un mensaje que se ha ejecutado. En una clase como la de Examen no tiene un sentido práctico el destructor.

En main, no hay que llamar a ningún método de manera explícita ya que estos métodos se caracterizan por ser llamados de manera automática.

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

int main(){
  Examen a;
  Examen b(1, 10, 2021);
  Examen c[5];
  a.mostrar();
  b.mostrar();
  cout << endl;
  return 0;
}
```

**Código 13** · Función main haciendo uso de la clase Examen.

La salida por pantalla de este programa sería algo como la siguiente:

```
,  - 0/0/0 - Calificación: 0
,  - 1/10/2021 - Calificación: 0
Llamado al destructor de examen
Llamado al destructor de examen
Llamado al destructor de examen
Llamado al destructor de examen
Llamado al destructor de examen
Llamado al destructor de examen
Llamado al destructor de examen
```

Se puede observar como el método mostrar de la variable *a* muestra todos sus cadenas vacías y sus valores enteros en cero. La variable *b* muestra los valores de fecha con los valores asignados a partir de los parámetros del constructor. Por último, se muestran los siete mensajes de los llamados al destructor. Uno para cada una de las variables simples (a y b) y cinco más para cada elemento del vector *c*.