Programación Orientada a Objetos

Parcial 2

Notas de curso

Dr. Ezequiel Arceo May

22 de abril de 2019

Índice

«Un lenguaje de programación es una forma de expresarnos»
— Ezequiel Arceo —

1. Concepto básicos

Los conceptos más básicos que debemos entender en la Programación Orientada a Objetos (POO) son:

- Clase
- Objeto
- Abstracción
- Encapsulación
- Herencia
- Polimorfismo

1.1. Clase

Una **clase** es una **abstracción** que hacemos de nuestra experiencia sensible. El ser humano tiende a agrupar seres o cosas (que son los *objetos*) con características similares en grupos o *clases*.

Clasificar es algo que usamos en nuestro día a día:

Figura 1: Existen muchas razas de perros; hay perros pequeños y grandes, perros de colores claros y de colores oscuros. Sin embargo, cuando vemos un perro lo distinguimos rápidamente, sabemos que "eso es un perro", ya lo hemos identificado dentro de un grupo de objetos al que llamamos "perro". Esto es, decimos que es un objeto *son de la clase* **Perro**.

1.2. Objeto

Un **objeto** es una **colección de Atributos y Métodos**, un objeto se deriva de una clase. Un objeto es un elemento de una clase. Un objeto pertenece a una clase.

Todo lo que nos rodea es un objeto, un perro, un ser humano, una mesa, una cámara, una computadora, un celular, etc.

Considere el siguiente ejemplo:

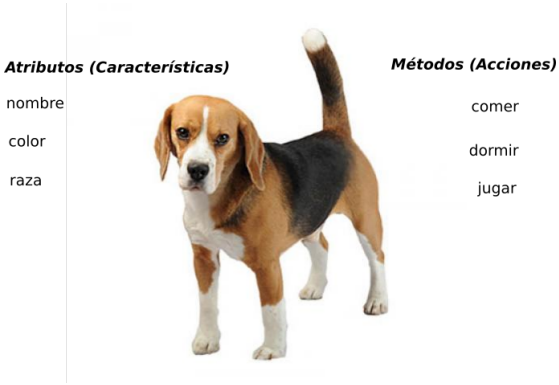


Figura 2: Un objeto perro de la clase **Perro** tiene Atributos como **nombre**, **raza**, **color**, etc, pero también tiene Métodos como **comer**, **dormir**, **jugar**, etc.

Todos los objetos tiene atributos (que son las características del objeto) y métodos (que son las acciones que el objeto realiza).

1.3. Abstracción

Una **abstracción** es el proceso mental de extraer los esencial de algo, ignorando los detalles irrelevantes/superfluos.

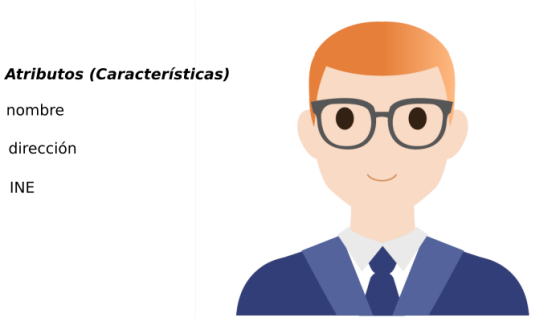


Figura 3: Si quisiéramos abstraer la clase **Persona**, tendríamos que considerar atributos relevantes como **nombre**, **dirección**, **INE**, etc. Ignoramos cosas tales como la cantidad de veces que ha visto su película favorita, el nombre de su perro, o la marca de sus calcetines.

1.4. Encapsulación

La **encapsulación** es un proceso mediante el cual ocultamos los detalles que dan soporte a las características de una abstracción.

Figura 4: Cuando vemos al médico y nos receta un medicamento en forma de píldora, esperamos que nos cure nuestra dolencia, no sabemos y no necesitamos saber qué es lo que contiene ni cómo funciona. La ciencia médica se ha encargado de poner lo esencial en la píldora, nosotros solamente la consumimos.

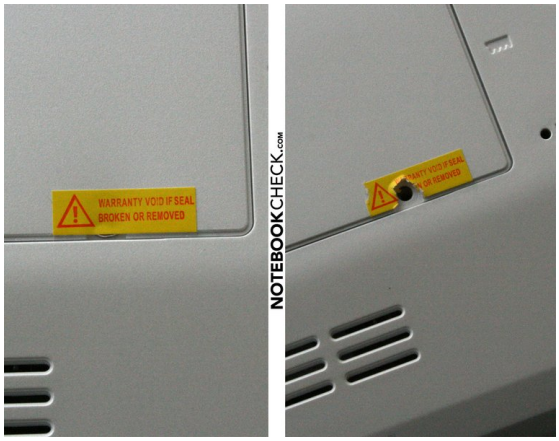


Figura 5: La encapsulación también es una medida de protección; si cualquiera tiene acceso al contenido interior puede ocasionar daños a propósito o accidentalmente. Recuerde los sellos de garantía en los productos que compra.

De forma práctica, la encapsulación aísla los atributos para que no los accedamos directamente.

1.5. Herencia

La **herencia** sucede cuando una clase nueva de crea a partir de una clase existente, obteniendo (heredando) todos sus atributos y métodos.

A la clase de la cual se hereda se le llama *clase padre* o *súper clase*. A la clase que recibe la herencia se le llama *clase hija* o *subclase*.

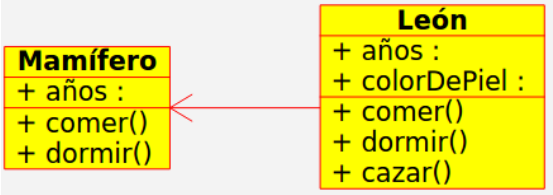


Figura 6: Ejemplo de herencia. **Mamífero** es la clase padre, y **León** es la clase hija. Note que todos los atributos y métodos de la clase **Mamífero** están en la clase **León**, junto con otros que son propios y especializan a la clase **León**.

El diagrama de la figura 6 se puede leer como:

- "la clase **León** hereda de la clase **Mamífero**",
- "la clase **León** es clase hija de la clase **Mamífero**",
- "la clase **León** es una subclase de la clase **Mamífero**",
- "la clase **Mamífero** es clase padre de la clase **León**",
- "la clase **Mamífero** es súper clase de la clase **León**".

Con la herencia nos ahorramos volver a definir los atributos y métodos de una clase. Basta con heredarlos de la clase padre y ya.

1.6. Polimorfismo

El **polimorfismo** es la cualidad que tiene los objetos de responder de distintas formas al mismo mensaje.

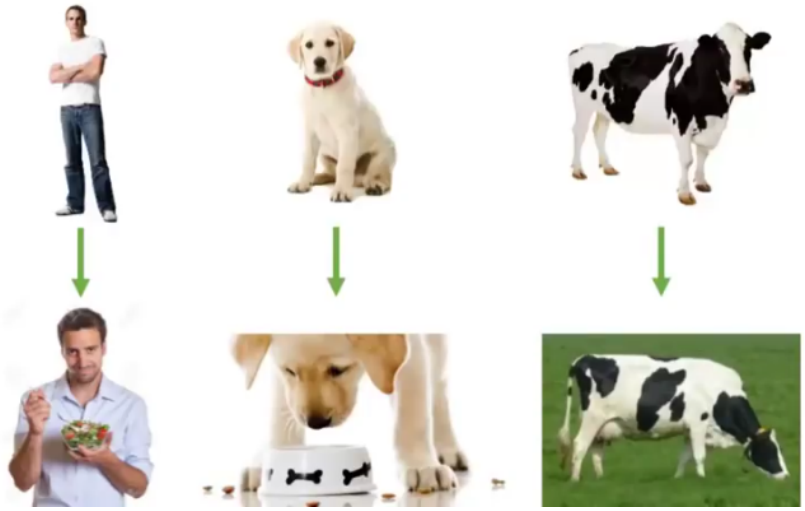


Figura 7: Considere objetos de las clases **Persona**, **Perro** y **Vaca**, cada uno de ellos puede **comer**, pero lo hacen de forma distinta.

2. Objetos en todas partes

Los objetos están en todas partes, es importante aprender a reconocerlos en situaciones de la vida real.

2.1. Reconoce objetos a partir de nombres

Imagine que tenemos que desarrollar una aplicación simple y recibimos una descripción con los requerimientos. La aplicación debe permitir al usuario calcular áreas y perímetros de cuadrados, rectángulos, círculos y elipses.

Entonces necesitamos calcular el área y perímetro de cuatro objetos. Los nombres que se refieren a estos objetos son:

- Cuadrado
- Rectangulo
- Circulo
- Elipse

Si tiene que dibujar estas figuras en papel para calcular manualmente su área y perímetro ¿qué información necesita para cada figura?

Figura	**Información requerida**
Cuadrado	lado
Rectangulo	base, altura
Circulo	radio
Elipse	semieje mayor, semieje menor

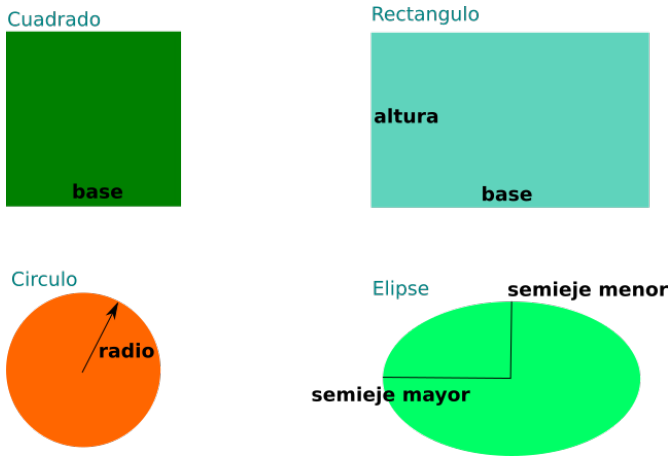


Figura 8: Figuras geométricas y su información requerida.

2.2. Haz plantillas para los objetos

Imagine ahora que tiene que calcular el área de cuatro rectángulos, cada uno con base y altura diferente. Terminaríamos con cuatro dibujos distintos, a menos que hagamos una plantilla en la cual solamente tengamos que especificar los valores de base y altura. A esta plantilla se le conoce como una **clase**. Una clase abstrae las características relevantes de los objetos.

Una vez que tenemos una clase, la podemos usar para crear objetos de dicha clase. A estos objetos se les llama **instancias** de clase.

2.3. Reconoce los atributos

Debemos estar seguros de que una clase tenga las variables necesarias que encapsulen toda la información requerida para realizar sus tareas.

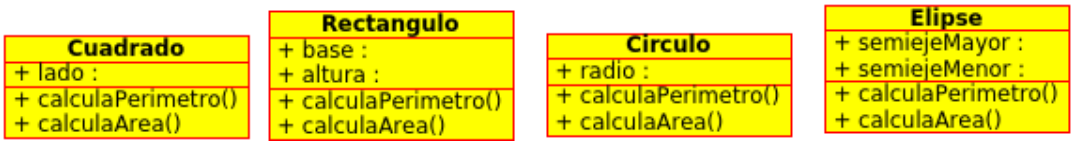
Nombre de Clase	**Atributos**
Cuadrado	lado
Rectangulo	base, altura
Circulo	radio
Elipse	semiejeMayor, semiejeMenor

La siguiente imagen muestra el diagram UML (Unified Modelling Language) muestra las cuatro clases con sus atributos

2.4. Reconoce acciones de los verbos

Hasta ahora hemos diseñado cuatro clases e identificado los atributos necesarios para cada una. Ahora, es tiempo de añadir las piezas necesarias de código que funciones con los atributos para desempeñar las tareas necesarias. En otras palabras, debemos encapsular dentro de las clases las funciones que procesen los atributos especificados en los objetos para realizar sus tareas.

En nuestro caso, para todas las figuras necesitamos calcular el perímetro y el área. Llamaremos `calculaPerimetro` a la función que calcula el perímetro de la figura *en función de sus atributos*. Llamaremos `calculaArea` a la función que calcula el área de la figura *en función de sus atributos*.



2.5. Organicemos las clases

Hasta ahora, nuestra solución orientada a objetos tiene cuatro clases, con sus atributos y métodos. Sin embargo, si nos damos cuenta, notaremos que las cuatro clases tienen los mismos métodos: `calculaPerimetro` y `calculaArea`. El código en los métodos de cada clase es diferente, porque cada figura usa datos y una fórmula diferente. Sin embargo, las declaraciones de estos métodos son idénticas: ambos tienen el mismo nombre, carecen de parámetros, y retornan un valor flotante.

Todas la formas tratadas son figuras geométricas. Podemos abstraer las características comunes, en este caso la capacidad de calcular el perímetro y área. Definimos así la clase **FiguraGeometrica** con los métodos **calculaPerimetro** y **calculaArea**, y hacemos que las otras cuatro clases hereden de esta:

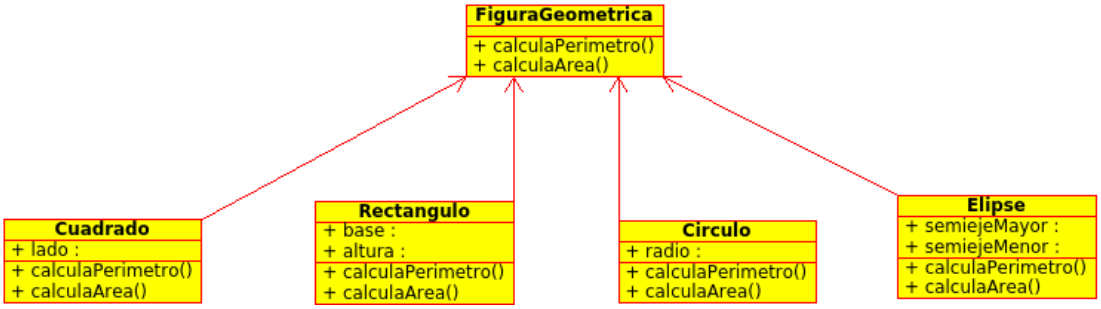


Figura 9: Jerarquía de figuras geométricas

Trabajo B1. Entrega Antes de: Lunes 25 de Marzo de 2019 a las 23:59.
Para cada clase en la figura 9 llene una tabla como la siguiente

Nombre	Atributos	Métodos	Clase Padre	Clase(s) Hija(s)

- Enviar por correo a `ezequiel_arceo@my.uvm.edu.mx`
- El asunto del correo será `POO_TrabajoB1_NOMBRE_DEL_AUTOR`
- El nombre de cada archivo adjunto será `POO_TrabajoB1_NOMBRE_DEL_AUTOR.*`

Ejemplo:
ASUNTO: `POO_TrabajoB1_JUAN_PEREZ_FERNANDEZ`
ADJUNTO: `POO_TrabajoB1_JUAN_PEREZ_FERNANDEZ.cpp`

3. Clases y Objetos en C++

Para hacer una clase en C++ usamos la palabra reservada `class`, seguida del nombre de la clase **con la primera letra en mayúscula**, un par de llaves {}, y finalizamos con un `;`.

3.1. Estructura de una clase

Dentro de las llaves, se escribe palabra reservada `private` seguida de `:` y se declaran los atributos de la clase. Se escribe la palabra reservada `public` seguida de `:` y declaran los métodos de la clase.

Siempre usaremos algo de la forma:

```
class NombreDeLaClase
{
  private: // Atributos
    // declaración del atributo 1
    // declaración del atributo 2
    // ...
  public: // Métodos
    // declaración del método 1
    // declaración del método 2
    // ...
}; // Es importante colocar ;
```

Todo lo declarado como `private` es accesible solamente a los elementos de la clase. Los elementos bajo `private` son una encapsulación de los datos de una clase.

Todo lo declarado como `public` es accesible desde el exterior de la clase. Esta es la interfaz de la clase.

Juntas, las declaraciones de atributos y métodos forman la abstracción de una clase. Podemos considerar que esta es la declaración de una clase, pero dicha clase todavía no puede ser usada porque no ha sido definida, es decir, no ha sido implementada.

3.2. Método Constructor

Toda clase debe definir al menos un método especial denominado **constructor**, con el mismo nombre de la clase, y que tome como argumentos a todos los atributos de la clase. Como su nombre lo indica, el constructor tiene la tarea de inicializar los atributos de un objeto de la clase con los valores de los parámetros que recibe al momento de la declaración.

Los constructores son especiales:

- tienen el mismo nombre que la clase a la que pertenecen.

- no tienen tipo de retorno, y por lo tanto no retornan ningún valor.
- no pueden ser heredados.
- deben ser públicos. No tendría ningún sentido declarar un constructor como privado, ya que siempre se usan desde el exterior de la clase, ni tampoco como protegido, ya que no puede ser heredado.

Por ejemplo, el constructor de la clase **Persona** con atributos **nombre** y **edad** se llama **Persona** y tiene dos argumentos, uno de tipo **string** y otro de tipo **int**.

Considere la siguiente abstracción simple de la clase **Persona**

```
1  //EJEMPLO: Declaración, implementación y uso de la clase Persona
2  #include <iostream>
3  #include <string>
4  using namespace std;
5
6  class Persona
7  {
8    private: // Atributos
9      string nombre;
10     int edad;
11   public: // Métodos
12     Persona(string,int); // Constructor (sin tipo de retorno!!)
13     void leer();
14     void caminar();
15 };
16
17 // Definición del método Constructor de la clase Persona
18 Persona::Persona(string _nombre, int _edad)
19 {
20   nombre = _nombre;
21   edad = _edad;
22 }
23 // Definición del método leer de la clase Persona
24 void Persona::leer()
25 {
26   cout<<"Soy "<<nombre<<" y estoy leyendo."<<endl;
27 }
28 // Definición del método caminar de la clase Persona
29 void Persona::caminar()
30 {
31   cout<<"Tengo "<<edad<<" años y estoy caminando."<<endl;
32 }
33 // continuará ...
```

IMPORTANTE: Dentro de la definición del método constructor **Persona** no hemos declarado las variables `nombre` y `edad`, estas variables son los atributos **nombre** y **edad** de la clase **Persona**. ¿Cómo es esto posible? . . . bueno, al usar `Persona::` especificamos que estamos trabajando en el ámbito de la clase **Persona** y por lo tanto, todos los miembros (atributos y métodos) de la clase **Persona** son accesibles.

Los métodos **leer** y **caminar** también tienen acceso a los atributos **nombre** y **edad** porque están en el ámbito de la clase **Persona**.

3.3. Creación de Objetos

Ahora que ya tenemos una clase podemos crear objetos, y usar sus métodos.

A la creación de objetos también se le denomina **instanciación de clase**, y se dice que el nuevo objeto es una instancia de la clase instanciada.

Para usar el método **miMetodo** de un objeto `miObjeto` escribimos `miObjeto.miMetodo(...)` donde `...` se refiere a la lista de argumentos que **miMetodo** requiere.

```
1  // ... continuación
2  //EJEMPLO: Declaración, implementación y uso de la clase Persona
3  void main()
4  {
5    // Definición de objetos de la clase Persona
6    // Forma 1 usando el constructor
7    Persona p1 = Persona("Ezequiel",33);
8    // Forma 2 usando el constructor
9    Persona p2("Itzel",6);
10
11   // Llamamos a los métodos de objetos Persona
12   p1.leer();
13   p2.caminar();
14 }
```

Un método de clase no es más que una función dentro del ámbito de la clase en donde se declara, por lo tanto puede requerir argumentos y puede tener valor de retorno, como cualquier función.

EJEMPLO: Construcción de una clase llamada **Rectangulo** con los atributos base y altura, y los métodos **perimetro** y **area**.

```cpp
//EJEMPLO: clase Rectangulo
#include <iostream>
#include <string>
using namespace std;

class Rectangulo
{
  private: // Atributos
    float base,altura;
  public: // Métodos
    Rectangulo(float,float); // Constructor (sin tipo de retorno!!)
    void perimetro();
    void area();
};
// Definición del método Constructor
Rectangulo::Rectangulo(float _base, float _altura)
{
  base = _base;
  altura = _altura;
}
void Rectangulo::perimetro()
{
  float _perimetro;
  _perimetro = 2*base + 2*altura;
  cout<<"El perimetro es "<<_perimetro<<" unidades."<<endl;
}
void Rectangulo::area()
{
  float _area;
  _area = base*altura;
  cout<<"El area es "<<_area<<" unidades cuadradas."<<endl;
}

void main()
{
  Rectangulo r1 = Rectangulo(5,10);
  r1.perimetro();
  r1.area();
  Rectangulo r2(2,3);
  r2.perimetro();
  r2.area();
}
```

Actividad 1.
Construya una clase llamada **Triangulo** con los atributos **base** y **altura**, y los métodos **perimetro** y **area**.
Trabajo B2. Entrega Antes de: Lunes 1 de Abril de 2019 a las 23:59.
Construya una clase llamada **Circulo** con el atributo **radio** y los métodos **perimetro**, y **area**.

- Enviar por correo a `ezequiel_arceo@my.uvm.edu.mx`
- El asunto del correo será `POO_TrabajoB2_NOMBRE_DEL_AUTOR`
- El nombre de cada archivo adjunto será `POO_TrabajoB2_NOMBRE_DEL_AUTOR.*`

Ejemplo:
ASUNTO: `POO_TrabajoB2_JUAN_PEREZ_FERNANDEZ`
ADJUNTO: `POO_TrabajoB2_JUAN_PEREZ_FERNANDEZ.cpp`

Trabajo B3. Entrega Antes de: Lunes 1 de Abril de 2019 a las 23:59.
Construya una clase llamada **Elipse** con los atributos **semiejeMayor** y **semiejeMenor**, y los métodos **perimetro** y **area**.

- Enviar por correo a `ezequiel_arceo@my.uvm.edu.mx`
- El asunto del correo será `POO_TrabajoB3_NOMBRE_DEL_AUTOR`
- El nombre de cada archivo adjunto será `POO_TrabajoB3_NOMBRE_DEL_AUTOR.*`

Ejemplo:
ASUNTO: `POO_TrabajoB3_JUAN_PEREZ_FERNANDEZ`
ADJUNTO: `POO_TrabajoB3_JUAN_PEREZ_FERNANDEZ.cpp`

4. Múltiples Constructores

Una clase puede tener uno o más constructores:

- un constructor muy común se llama **constructor por defecto**: se usa para instanciar una clase, sin especificar los parámetros en el constructor.
- los constructores que no son el constructor estándar o el constructor por defecto se llaman *constructores especializados*: estos son útiles cuando es necesario usar un formato de entrada de datos distinto al del constructor estándar.

El constructor por defecto y cualquier constructor especializado tienen el mismo nombre del constructor, es decir, se llaman igual que la clase. ¿Cómo es esto posible?

4.1. Sobrecarga de Funciones

C++ permite definir varias funciones con el mismo nombre dentro del mismo ámbito, siempre y cuando todas tengan una cantidad diferente de parámetros, o estos sean de distintos tipos. A esto se le denomina **sobrecarga de funciones**. Cuando las funciones "sobrecargadas" son constructores, decimos que hay **sobrecarga de constructores**.

```
1  // EJEMPLO: Sobrecarga de funciones
2  #include <iostream>
3  using namespace std;
4  // Función cuadrado para valores int
5  int cuadrado(int x)
6  {
7    int _cuadrado = x*x
8    cout<<"El cuadrado del int "<<x<<" es "<<_cuadrado<<endl;
9    return cuadrado;
10 }
11 // Función cuadrado para valores float
12 float cuadrado(float x)
13 {
14   float _cuadrado = x*x;
15   cout<<"El cuadrado del float "<<x<<" es "<<_cuadrado<<endl;
16   return cuadrado;
17 }
18
19 void main()
20 {
21   cuadrado(3);
22   cuadrado(3.0);
23 }
```

Actividad 2.
Sobrecargue una función llamada `suma` que sume dos números. Considere que los argumentos pueden ser de tipo **int** o **float**. El tipo de retorno de todas será **float**. HINT: Primero determine cuantas funciones necesita escribir.

4.2. Constructor por defecto

El constructor por defecto permite dar valores por defecto a los atributos de un objeto. El constructor por defecto no tiene argumentos.

```
// EJEMPLO: Constructor por defecto (Sobrecarga de funciones)
#include <iostream>
using namespace std;
class Punto
{
  private: // Atributos
    float x,y;
  public:  // Métodos
    Punto(float,float); // Constructor
    Punto();            // Constructor por defecto
    void mostrarPunto();
};
// Constructor
Punto::Punto(float _x, float _y)
{// Colocamos el punto en donde indica el usuario
  x = _x;
  y = _y;
}
// Constructor por defecto
Punto::Punto()
{// Colocamos al punto en el origen
  x = 0.0;
  y = 0.0;
}
void Punto::mostrarPunto()
{
  cout<<"("<<x<<", "<<y<<")"<<endl;
}

void main()
{
   Punto p1(2.0,3.0);
   Punto p2;

   p1.mostrarPunto();
   p2.mostrarPunto();
}
```

Al usar constructores por defecto se emula el comportamiento de tipos estándar como **int**, **float** o **string** que dan valores por defecto a las variables al momento de ser declaradas:

```cpp
// EJEMPLOS: valores por defecto en tipos estándar
#include <iostream>
#include <string>
using namespace std;

int main()
{
  int entero;
  float flotante;
  string cadena;
  cout << "Valor int por defecto '" << entero <<"'"<< endl;
  cout << "Valor float por defecto '" << flotante <<"'"<< endl;
  cout << "Valor strng por defecto '" << cadena <<"'"<< endl;
}
```

4.3. Constructores especializados

Los constructores especializados permiten instanciar objetos aceptando como parámetros un subconjunto de los atributos de la clase, o también permiten aceptar formatos específicos de entrada de datos.

Considere por ejemplo la clase **Fecha** cuyo constructor (estándar) acepta tres enteros, el año, el mes y el día. El método **mostrarFecha** imprime la fecha en el formato `DD/MM/YYYY` por ejemplo, `28/03/2019`. Pero es posible que esta misma fecha se escriba como `20190328`. Podemos usar un constructor especializado para inicializar una fecha con el formato `YYYYMMDD`

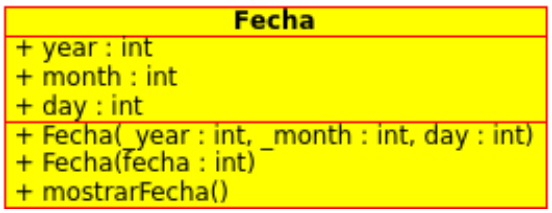


Figura 10: Clase **Fecha** con constructor especializado

```cpp
// EJEMPLO: Constructor especializado
#include <iostream>
using namespace std;
class Fecha
{
  private: // Atributos
    int year;
    int month;
    int day;
  public: // Métodos
    Fecha(int,int,int); // Constructor (estándar)
    Fecha(int); // Constructor especializado, del formato YYYYMMDD
    void mostrarFecha();
};
// Constructor (estándar)
Fecha::Fecha(int _year,int _month,int _day)
{
  year =_year;
  month = _month;
  day = _day;
}
void Fecha::mostrarFecha()
{
  cout<<day<<"/"<<month<<"/"<<year<<endl;
}
// Constructor especializado
Fecha::Fecha(int fecha)
{// Acepta fechas en formato YYYYMMDD
  year = fecha / 10000;
  fecha = fecha % 10000;
  month = fecha / 100;
  fecha = fecha % 100;
  day = fecha;
}
int main()
{
  Fecha hoy = Fecha(2019,03,28);
  Fecha ayer = Fecha(20190327);
  hoy.mostrarFecha();
  ayer.mostrarFecha();
  return 0;
}
```

Trabajo B4. Entrega Antes de: Lunes 1 de Abril de 2019 a las 23:59.
Cree la clase **Tiempo** como se muestra en el siguiente diagrama:

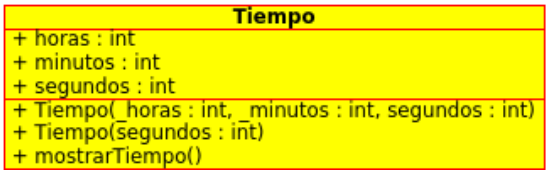


Figura 11: Clase **Tiempo** con constructor especializado

El método **mostrarTiempo()** imprime la hora en formato `horas:minutos:segundos`. El método **Tiempo(int)** acepta la hora dada en segundos. Por ejemplo, la 1:20 AM se escribiría en segundos como 4800.

- Enviar por correo a `ezequiel_arceo@my.uvm.edu.mx`
- El asunto del correo será `POO_TrabajoB4_NOMBRE_DEL_AUTOR`
- El nombre de cada archivo adjunto será `POO_TrabajoB4_NOMBRE_DEL_AUTOR.*`

Ejemplo:
ASUNTO: `POO_TrabajoB4_JUAN_PEREZ_FERNANDEZ`
ADJUNTO: `POO_TrabajoB4_JUAN_PEREZ_FERNANDEZ.cpp`

5. Método Destructor

Un objeto inicia su *ciclo de vida* al ser instanciado por el método Constructor de su clase. Este ciclo termina con otro método especial, llamado **método Destructor**. El método destructor sirve para eliminar un objeto de una clase determinada.

Para cualquier clase, el método destructor:

- se llama igual que la clase pero se antepone el símbolo ~. Por ejemplo, el destructor de la clase **Punto** se llama **~Punto**.
- no tiene tipo,
- no tiene parámetros,
- no puede ser heredado,
- no puede ser sobrecargado (obviamente, porque no tiene tipo ni argumentos).

El método destructor de un objeto es llamado automáticamente cuando este sale de su ámbito. Esto es así, excepto cuando el objeto fue creado dinámicamente con el operador `new`, en tal caso, cuando es necesario eliminarlo, hay que hacerlo explícitamente usando el operador `delete`.

A continuación se presenta un ejemplo de definición del método destructor para una clase llamada **Perro**.

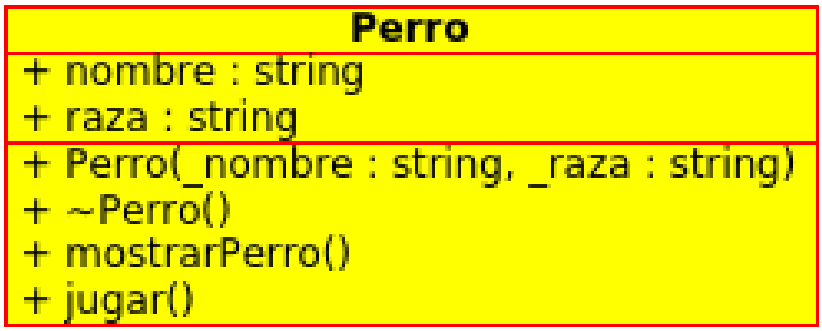


Figura 12: El **método destructor** tiene el mismo nombre que la clase a la que pertenece, pero lleva el prefijo ~.

```cpp
// EJEMPLO: Metodo Destructor
#include <iostream>
#include <string>
using namespace std;
class Perro
{
  private:
    string nombre, raza;
  public:
    Perro(string,string); // Constructor
    ~Perro(); // Destructor
    void mostrarPerro();
    void jugar();
};
// Constructor
Perro::Perro(string _nombre,string _raza)
{
  nombre = _nombre;
  raza = _raza;
}
// Destructor
Perro::~Perro()
{
  // Esto es todo, no tenemos que colocar nada aquí.
}
void Perro::mostrarPerro()
{
  cout<<"Nombre: "<<nombre<<endl;
  cout<<"Raza: "<<raza<<endl;
}
void Perro::jugar()
{
  cout<<"El perro "<<nombre<<" está jugando."<<endl;
}

int main()
{
  Perro p1 = Perro("Tobi","chihuahua");
  p1.mostrarPerro();
  p1.jugar();
  p1.~Perro(); // Destruye al objeto p1
  return 0;
}
```

6. Getters & Setters

En la declaración de una clase, el *modificador de acceso* `private` hace que los atributos solamente sean accesibles desde los métodos (públicos) de la misma clase. Los atributos están encapsulados.

A pesar del encapsulamiento, necesitamos una forma acceder a los atributos, ya sea porque requerimos conocer sus valores, o porque queremos cambiar dichos valores. Para esto usamos los métodos **getters** y los métodos **setters**.

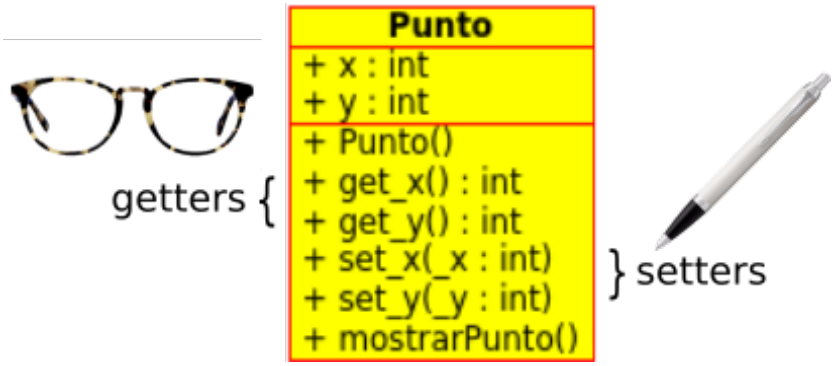


Figura 13: Los métodos **getters** nos sirven para obtener (leer) los valores de los atributos de un objeto. Los métodos **setters** nos sirven para cambiar (escribir) los valores de los atributos de un objeto.

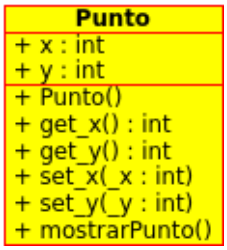
Es práctica común que el nombre de un **método getter** se construya anteponiendo el prefijo `get` o `get_` al nombre del atributo que **lee**.

Es práctica común que el nombre de un **método setter** se construya anteponiendo el prefijo `set` o `set_` al nombre del atributo que **escribe**.

En general, los métodos getters son del mismo tipo que el atributo que leen (porque devuelven el valor actual de dicho atributo). Los métodos setters son en general de tipo **void** (a menos que se requiera devolver información sobre el éxito/fallo del cambio de valor).

Podemos usar a los métodos getters y setters como centinelas de los atributos, permitiéndonos reaccionar ante los intentos de lectura y escritura.

Lea con detenimiento la implementación de la clase **Punto**. Los métodos getters y setters en ella imprimen información cada vez que son llamados.



```cpp
// EJEMPLO: Métodos Getters y Setters
#include <iostream>
using namespace std;
class Punto
{
  private: // Atributos
    int x, y;
  public: // Métodos
    Punto(); // Constructor por defecto
    int get_x(); // getter de x
    int get_x(); // getter de y
    void set_x(int _x); // setter de x
    void set_y(int _y); // detter de y
    void mostrarPunto();
};
// Constructor por defecto
Punto::Punto()
{
}
// Getters
int Punto::get_x()
{
  cout << "Alguien lee a 'x'" << endl;
  return x;
}
int Punto::get_y()
{
  cout << "Alguien lee a 'y'" << endl;
  return y;
}
// Setters
void Punto::set_x(int _x)
{
  cout << "Alguien asigna a 'x' el valor " << _x << endl;
  x = _x;
}
void Punto::set_y(int _y)
{
  cout << "Alguien asigna a 'y' el valor " << _y << endl;
  y = _y;
}
```

```cpp
// EJEMPLO: Métodos Getters y Setters (continuación)
void Punto::mostrarPunto()
{
  cout << "(" << x << ", " << y << ")" << endl;
}

int main()
{
  Punto p1 = Punto();
  p1.setX(2);
  p1.setY(4);
  p1.mostrarPunto();
  p1.setY(2);
  p1.mostrarPunto();
  return 0;
}
```

Actividad 3.
Construya una clase llamada **Triangulo** con los atributos **base** y **altura**, y los métodos **perimetro** y **area**. La clase debe tener métodos getters y setters para cada atributo.

Trabajo B5. Entrega Antes de: Lunes 1 de Abril de 2019 a las 23:59.
Construya una clase llamada **Rectangulo** con los atributos **base**, **altura** y **color** ,y los métodos **perimetro**, **area** y **mostrarRectangulo**. La clase debe tener métodos getters y setters para cada atributo.

- **perimetro** calcula e imprime el perímetro del rectángulo.
- **area** calcula e imprime el área del rectángulo.
- **mostrarRectangulo** imprime los atributos del rectángulo.

- Enviar por correo a `ezequiel_arceo@my.uvm.edu.mx`
- El asunto del correo será `POO_TrabajoB5_NOMBRE_DEL_AUTOR`
- El nombre de cada archivo adjunto será `POO_TrabajoB5_NOMBRE_DEL_AUTOR.*`

Ejemplo:
ASUNTO: `POO_TrabajoB5_JUAN_PEREZ_FERNANDEZ`
ADJUNTO: `POO_TrabajoB5_JUAN_PEREZ_FERNANDEZ.cpp`

Trabajo B6. Entrega Antes de: Lunes 8 de Abril de 2019 a las 23:59.
A partir del programa `gato-pp.cpp` explicado en la clase del día **Jueves 4 de Abril de 2019** escriba una clase llamada **Gato** cuyos atributos sean las variables del programa y cuyos métodos sean las funciones del programa

En la función `main` de su código se debe crear una variable de tipo **Gato** y llamar a sus métodos para hacer que el juego funcione.

(NOTA: la función `main` del archivo `gato-pp.cpp` NO es un método).

Puede descargar el archivo `gato-pp.cpp` en el siguiente enlace:
`gato-pp.cpp`

- Enviar por correo a `ezequiel_arceo@my.uvm.edu.mx`
- El asunto del correo será `POO_TrabajoB6_NOMBRE_DEL_AUTOR`
- El nombre de cada archivo adjunto será `POO_TrabajoB6_NOMBRE_DEL_AUTOR.*`

Ejemplo:
ASUNTO: `POO_TrabajoB6_JUAN_PEREZ_FERNANDEZ`
ADJUNTO: `POO_TrabajoB6_JUAN_PEREZ_FERNANDEZ.cpp`

7. Interrupciones durante un ciclo

Es conveniente poder dar por terminado un ciclo o interrumpir/saltar una de sus iteraciones:

- la palabra reservada `break` hace que un ciclo termine abruptamente.
- la palabra reservada `continue` hace que un ciclo salte hasta el final del cuerpo y pase a la siguiente iteración.

El siguiente programa imprime el valor de un contador entero, mientras este tiene un valor distinto de 5, entonces termina.

```
1  // EJEMPLO: break
2  #include <iostream>
3
4  using namespace std;
5  void main()
6  {
7    for (int i = 0; i < 10; i++)
8    {
9      if( i == 5 )
10     {
11       break;
12     }
13     cout << "El valor de i es " << i << endl;
14   }
15 }
```

El siguiente programa imprime los valores impares de un contador entero.

```
1  // EJEMPLO: continue
2  #include <iostream>
3
4  using namespace std;
5  void main()
6  {
7    for (int i = 0; i < 10; i++)
8    {
9      if( i % 2 == 0 )
10     {
11       continue;
12     }
13     cout << "El valor de i es " << i << endl;
14   }
15 }
```

En el siguiente ejemplo vemos como emplear `break` para determinar si un elemento está es un vector. La razón para usar `break` es ahorrar tiempo y capacidad de cómputo. Si ya encontré un elemento dentro de un vector, ya no tengo que seguir buscando.

```cpp
// EJEMPLO: break
#include <iostream>
#include <vector>
using namespace std;

bool encuentraEnVector(vector<int> vec, int elemento)
{
  bool encontrado = false;
  for(int i=0; i < vec.size(); i++)
  {
    if (vec[i] == elemento)
    {
      encontrado = true;
      break;
    }
  }
  return encontrado;
}
```

Usando la función `encuentraEnVector` podemos hacer una función que halle los N números mayores en un vector de enteros:

```cpp
// EJEMPLO: continue
#include <iostream>
#include <vector>
using namespace std;

vector<int> indicesDeMaximos(vector<int> vec, int cantidadDeMaximos)
{
  vector<int> indices; // indices de los máximos en vec
  while( indices.size() < cantidadDeMaximos )
  {
    int indiceDeMaximo = 0;
    int valorDeMaximo = vec[0];
    for(int i=0; i < vec.size(); i++)
    {
      if ( encuentraEnVector(indices, i) )
      {
        continue;
      }
      if ( vec[i] > valorDeMaximo )
      {
        valorDeMaximo = vec[i];
        indiceDeMaximo = i;
      }
    }
    indices.push_back(indiceDeMaximo);
  }
  return indices;
}

int main()
{
  vector<int> numeros = {2,3,5,2,12,26,3,9,1};
  vector<int> indices = indicesDeMaximos(numeros,2);
  cout << "Los máximos se encuentran en las posiciones:" << endl;
  for (int i=0; i< indices.size(); i++)
  {
    cout << indices[i] << endl;
  }
  return 0;
}
```

Actividad 4.
Construya una función que halle los N números menores en un vector de enteros.