



# Programación II

---



# Programación Orientada a Objetos

---

## ■ Introducción

El término de Programación Orientada a Objetos indica más una forma de diseño y una metodología de desarrollo de software que un lenguaje de programación, ya que en realidad se puede aplicar el Diseño Orientado a Objetos (En inglés abreviado OOD, Object Oriented Design), a cualquier tipo de lenguaje de programación.

El desarrollo de la OOP empieza a destacar durante la década de lo 80 tomando en cuenta la programación estructurada, a la que engloba y dotando al programador de nuevos elementos para el análisis y desarrollo de software.

Básicamente la OOP permite a los programadores escribir software, de forma que esté organizado en la misma manera que el problema que trata de modelar.



# Programación Orientada a Objetos

---

Los lenguajes de programación convencionales son poco más que una lista de acciones a realizar sobre un conjunto de datos en una determinada secuencia. Si en algún punto del programa modificamos la estructura de los datos o la acción realizada sobre ellos, el programa cambia.

La OOP aporta un enfoque nuevo, convirtiendo la estructura de datos en el centro sobre el que pivotan las operaciones. De esta forma, cualquier modificación de la estructura de datos tiene efecto inmediato sobre las acciones a realizar sobre ella, siendo esta una de la diferencias radicales respecto a la programación estructurada.

Para quienes no están familiarizados con la programación estructurada diremos. En esta forma de diseño se descomponen los requerimientos del programa paso a paso, hasta llegar a un nivel que permite expresarlos mediante procedimientos y funciones. La OOP estructura los datos en objetos que pueden almacenar, manipular y combinar información.



# Programación Orientada a Objetos

---

En resumen, la programación estructurada presta atención al conjunto de acciones que manipulan el flujo de datos (desde la situación inicial a la final), mientras que la programación orientada a objetos presta atención a la interrelación que existe entre los datos y las acciones a realizar con ellos.

Muchos habrán oído comentarios sobre la incidencia de la OOP sobre la programación convencional. Se ha llegado a decir que el cambio introducido por la OOP es similar al producido por la aparición del ensamblador sobre el código de máquina.

La OOP proporciona las siguientes ventajas sobre otros lenguajes de programación



# Programación Orientada a Objetos

---

- **Uniformidad.** Ya que la representación de los objetos lleva implícita tanto el análisis como el diseño y la codificación de los mismos.
- **Comprensión.** Tanto los datos que componen los objetos, como los procedimientos que los manipulan, están agrupados en clases, que se corresponden con las estructuras de información que el programa trata.
- **Flexibilidad.** Al tener relacionados los procedimientos que manipulan los datos con los datos a tratar, cualquier cambio que se realice sobre ellos quedará reflejado automáticamente en cualquier lugar donde estos datos aparezcan.
- **Estabilidad.** Dado que permite un tratamiento diferenciado de aquellos objetos que permanecen constantes en el tiempo sobre aquellos que cambian con frecuencia permite aislar las partes del programa que permanecen inalterables en el tiempo.



# Programación Orientada a Objetos

---

- **Reusabilidad.** La noción de objeto permite que programas que traten las mismas estructuras de información reutilicen las definiciones de objetos empleadas en otros programas e incluso los procedimientos que los manipulan. De esta forma, el desarrollo de un programa puede llegar a ser una simple combinación de objetos ya definidos donde estos están relacionados de una manera particular

Uno de los puntos clave a remarcar en esta introducción es que la programación orientada a objetos no sustituye a ninguna metodología ni lenguaje de programación anterior. Todos los programas que se realizan según OOD se pueden realizar igualmente mediante programación estructurada. Su uso en la actualidad se justifica porque el desarrollo de todas las nuevas herramientas basadas en un interface de usuario gráfico como Windows, OS/2, x-Windows, etc. Es mucho más sencillo.



# Programación Orientada a Objetos - Definiciones

---

Durante el curso iremos introduciendo nuevos conceptos que normalmente se asocian a la programación orientada a objetos, como son: objeto, mensaje, método, clase, herencia, interfaz, etc.

- **Objeto:**

Un objeto es una unidad que engloba en sí mismo datos y procedimientos necesarios para el tratamiento de esos datos. Hasta ahora habíamos hecho programas en los que los datos y las funciones estaban perfectamente separadas, cuando se programa con objetos esto no es así, cada objeto contiene datos y funciones. Y un programa se construye como un conjunto de objetos, o incluso como un único objeto.

- **Mensaje:**

El mensaje es el modo en que se comunican los objetos entre si. En C++, un mensaje no es más que una llamada a una función de un determinado objeto. Cuando llamemos a una función de un objeto, muy a menudo diremos que estamos enviando un mensaje a ese objeto.

En este sentido, mensaje es el término adecuado cuando hablamos de programación orientada a objetos en general.



# Programación Orientada a Objetos - Definiciones

---

- **Método:**

Se trata de otro concepto de POO, los mensajes que lleguen a un objeto se procesarán ejecutando un determinado método. En C++ un método no es otra cosa que una función o procedimiento perteneciente a un objeto.

- **Clase:**

Una clase se puede considerar como un patrón para construir objetos. En C++, un objeto es sólo un tipo de variable de una clase determinada. Es importante distinguir entre objetos y clases, la clase es simplemente una declaración, no tiene asociado ningún objeto, de modo que no puede recibir mensajes ni procesarlos, esto únicamente lo hacen los objetos.

- **Interfaz:**

Las clases y por lo tanto también los objetos, tienen partes públicas y partes privadas. Algunas veces llamaremos a la parte pública de un objeto su interfaz. Se trata de la única parte del objeto que es visible para el resto de los objetos, de modo que es lo único de lo que se dispone para comunicarse con ellos.





# Programación Orientada a Objetos - Definiciones

---

- **Herencia:**

Veremos que es posible diseñar nuevas clases basándose en clases ya existentes. En C++ esto se llama derivación de clases, y en POO herencia. Cuando se deriva una clase de otra, normalmente se añadirán nuevos métodos y datos. Es posible que algunos de estos métodos o datos de la clase original no sean válidos, en ese caso pueden ser enmascarados en la nueva clase o simplemente eliminados. El conjunto de datos y métodos que sobreviven, es lo que se conoce como herencia.



# Clases

---

Una **clase** es una plantilla que se utiliza para crear múltiples objetos con características similares.

Las clases engloban todas las características de un conjunto particular de objetos. Cuando se escribe un programa en un lenguaje orientado a objetos, no se definen objetos individuales, sino que se definen clases de objetos

## **Declaración de una clase**

```
class <identificador de clase> [<:lista de clases base>] {  
    <lista de miembros>  
} [<lista de objetos>;
```



# Programación Orientada a Objetos

---

Las propiedades de cada clase deben cumplir una serie de premisas

- Las propiedades deben ser significativas dentro del entorno de la aplicación es decir, deben servir para identificar claramente y de una manera única (y unívoca) a cada uno de los objetos.
- El número de propiedades de un objeto debe ser el mínimo para realizar todas las operaciones que requiera la aplicación.

Definamos una clase rectángulo. Esta clase puede tener como atributos un punto (x,y), la anchura (a) y la longitud (l). Las operaciones a realizar son: mover, agrandar, reducir, etc. ¿Es posible realizarlas con las propiedades de la clase?

Un análisis posterior nos indica que es posible la realización de estas operaciones con los atributos definidos. Pero si incluimos la operación girar, vemos que con las propiedades definidas para la clase esta operación no se puede realizar. Para incluir esta nueva operación debemos redefinir las propiedades del objeto, en este caso las coordenadas de los vértices.



# Programación Orientada a Objetos

---

La lista de clases base se usa para derivar clases, de momento no le prestes demasiada atención, ya que por ahora sólo declararemos clases base.

La lista de miembros será en general una lista de funciones y datos.

Los datos se declaran del mismo modo en que lo hacíamos hasta ahora, salvo que no pueden ser inicializados, recuerda que estamos hablando de declaraciones de clases y no de definiciones de objetos.

Las funciones pueden ser simplemente declaraciones de prototipos, que se deben definir aparte de la clase pueden ser también definiciones.

Cuando se definen fuera de la clase se debe usar el operador de ámbito "::".



# Clases

---

## Ejemplo:

```
class circulo {  
    private:  
        float radio;  
    public:  
        void cambiar_radio(float r);  
        float obtener_radio();  
        float obtener_permetro();  
        float obtener_area();  
};
```



# Clases

---

## Definición de miembros de una clase

```
void circulo::cambiar_radio(float r){  
    radio=r;  
}
```

```
float circulo::obtener_radio(){  
    return radio;  
}
```

```
float circulo::obtener_perimetro(){  
    return 2*3.1416*radio;  
}
```

```
float circulo::obtener_area(){  
    return 3.1416*radio*radio;  
}
```



# Clases

---

## Crear un Objetos

```
#include <iostream.h>
void main(){
    circulo tapadera;
        tapadera.cambiar_radio(5);
        cout << "El Perimetro de la Tapadera es:" <<
            tapadera.obtener_perimetro();

        cout << "El Area de la Tapadera es:" <<
            tapadera.obtener_area();

}
```



# Clases

---

## **Especificadores de acceso:**

Dentro de la lista de miembros, cada miembro puede tener diferentes niveles de acceso.

En nuestro ejemplo hemos usado dos de esos niveles, el privado y el público, aunque hay más.

```
class <identificador de clase> {  
    public:  
        <lista de miembros>  
    private:  
        <lista de miembros>  
    protected:  
        <lista de miembros>  
};
```





# Clases

---

- **Acceso privado, private:**

Los miembros privados de una clase sólo son accesibles por los propios miembros de la clase y en general por objetos de la misma clase, pero no desde funciones externas o desde funciones de clases derivadas.

- **Acceso público, public:**

Cualquier miembro público de una clase es accesible desde cualquier parte donde sea accesible el propio objeto.

- **Acceso protegido, protected:**

Con respecto a las funciones externas, es equivalente al acceso privado, pero con respecto a las clases derivadas se comporta como público.

Cada una de éstas palabras, seguidas de ":", da comienzo a una sección, que terminará cuando se inicie la sección siguiente o cuando termine la declaración de la clase. Es posible tener varias secciones de cada tipo dentro de una clase.

Si no se especifica nada, por defecto, los miembros de una clase son privados.



# Clases

---

- ***Constructores***

Los constructores son funciones miembro especiales que sirven para inicializar un objeto de una determinada clase al mismo tiempo que se declara.

Los constructores tienen el mismo nombre que la clase, puede haber mas de uno por clase, no retornan ningún valor y no pueden ser heredados. Además 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.



# Clases

---

## **Ejemplo de clase circulo con un constructor:**

```
class circulo {  
    private:  
        int radio;  
    public:  
        circulo();  
        circulo(float r);  
        void cambiar_radio(float r);  
        float obtener_radio();  
        float obtener_permetro();  
        float obtener_area();  
};
```



# Clases

---

## **Definición de los constructores:**

```
circulo::circulo(){  
    radio=0;  
}
```

```
Circulo::circulo(float r){  
    radio=r;  
}
```



# Clases

---

Si una clase posee constructor, será llamado siempre que se declare un objeto de esa clase, y si requiere argumentos, es obligatorio suministrarlos.

ejemplo:

```
#include <iostream.h>
void main(){
    circulo tapadera(5);
    circulo tapon;
    tapon.cambiar_radio(1);

    cout << "El Perimetro de la Tapadera es:" <<
        tapadera.obtener_perimetro();

    cout << "El Area de la Tapadera es:" <<
        tapadera.obtener_area();

    cout << "El Perimetro de la Tapon es:" <<
        tapon.obtener_perimetro();

    cout << "El Area de la Tapon es:" <<
        tapon.obtener_area();
}
```



# Clases

---

## **Definición de los destructores:**

- Los destructores son funciones miembro especiales que sirven para eliminar un objeto de una determinada clase, liberando la memoria utilizada por dicho objeto.
- Los destructores tienen el mismo nombre que la clase, pero con el símbolo ~ delante, no retornan ningún valor y no pueden ser heredados.
- Cuando se define un destructor para una clase, éste es llamado automáticamente cuando se abandona el ámbito en el que fue definido. Esto es así salvo cuando el objeto fue creado dinámicamente con el operador new, ya que en ese caso, si es necesario eliminarlo, hay que usar el operador delete.
- En general, será necesario definir un destructor cuando nuestra clase tenga datos miembro de tipo puntero, aunque esto no es una regla estricta. El destructor no puede sobrecargarse, por la sencilla razón de que no admite argumentos.



# Clases

---

## Ejemplo de un destructor:

```
class circulo {  
    private:  
        int radio;  
    public:  
        circulo();  
        circulo(float r);  
        ~circulo();  
        void cambiar_radio(float r);  
        float obtener_radio();  
        float obtener_permetro();  
        float obtener_area();  
};  
  
circulo::~~circulo(){  
    cout << "Adios...";  
}
```