

I102 - Paradigmas de Programación

PROGRAMACIÓN ORIENTADA A OBJETOS

4



Prof. Mariano Scaramal
Ingeniería en Inteligencia Artificial

Que conceptos de C++?

- Conceptos:
 - Structs in C++
 - Conceptos de Programación Orientada a Objetos (OOP)
 - Ejemplos de Implementación de Clases en C++
 - El Archivo main, header y source.
 - Las Palabras Reservadas *private* y *public*.
 - Constructores y Destructores
 - El Puntero *this*

Structures en C++

- En C++, las structures (también llamadas structs) poseen mayor funcionalidad que en C (ver cuadro).
- Un struct de C++ permite el uso de OOP como métodos, herencia, constructores, etc.
- En C, un struct se usa para agrupar datos y crear estructuras de datos abstractas.
- En C++ se recomienda el uso de *using* en vez de *typedef* para definir alias.

Característica	Struct en C	Struct en C++
Nivel de acceso default	public	public
Funciones miembro	No incluido	Incluido
Herencia	No incluido	Incluido
Constructores/destructores	No incluido	Incluido
Overloading de operator()	No incluido	Incluido
Templates	No incluido	Incluido
Namespaces	No incluido	Incluido

Structures en C++ - Ejemplos

El struct modela un rectángulo con sus atributos length y width, con los cuales puede usar para calcular (una acción) su área y perímetro.

```
#include <iostream>

struct Rectangle {
    // Variables
    int length, width;
    // Funcion miembro que calcula el area
    int area() {
        return length * width;
    }
    // Funcion miembro que calcula el perimetro
    int perimetro() {
        return 2 * (length + width);
    }
};
```

```
int main() {
    Rectangle rect;
    rect.length = 5;
    rect.width = 3;

    // Muestra el area y el perimetro del rectangulo
    std::cout << "Area: " << rect.area()
    << std::endl;    // Imprime: 15
    std::cout << "Perimetro: " << rect.perimetro()
    << std::endl;    // Imprime: 16

    return 0;
}
```

Structures en C++ - Using

Si bien *using* se puede usar con templates (próximo tema) y typedef no, el uso de *using* es más legible que *typedef*, sólo por esos motivos es preferible usar *using*.

```
typedef struct rectangle{  
    // Variables  
    int length, width;  
    // Funcion miembro que calcula el area  
    int area() { return length * width; }  
    // Funcion miembro que calcula el perimetro  
    int perimetro() { return 2 * (length + width); }  
} Rectangle;
```

```
int main() {  
    Rectangle rect;  
    :  
}
```

Aquí, existe rectangle, el nombre dado al struct creado, y Rectangle, el alias dado con *typedef*.

```
using Rectangle = struct {  
    // Variables  
    int length, width;  
    // Funcion miembro que calcula el area  
    int area() { return length * width; }  
    // Funcion miembro que calcula el perimetro  
    int perimetro() { return 2 * (length + width); }  
};
```

```
int main() {  
    Rectangle rect;  
    :  
}
```

Aquí, *using* define directamente el alias para un struct determinado.

OOP Vs Programación Procedural

Los **lenguajes procedurales** (como C) están orientados a que el programador indique cada una de sus acciones, las cuales pueden ser agrupadas en funciones o procedimientos

- Las funciones y procedimientos son las unidades de programación
- Los datos poseen un scope global o de una función.

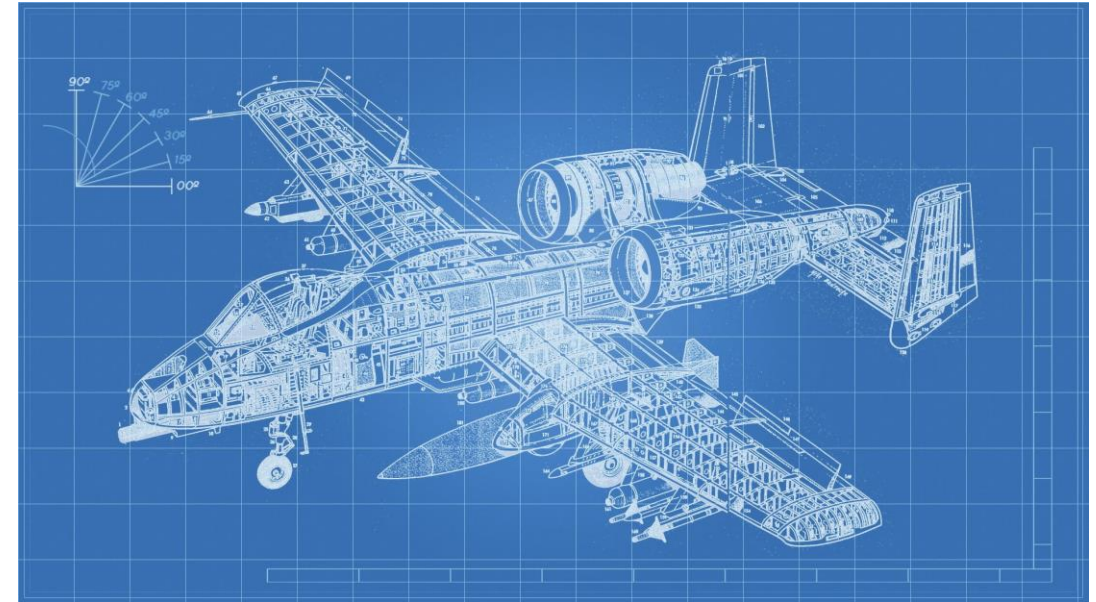
En el caso de C++, este es un lenguaje multi-paradigma que incluya la programación orientado a objetos (o en inglés, OOP) en donde:

- Los datos y la información están encapsulados en paquetes llamados clases.
- Las clases son la unidad de programación, se utiliza para crear objetos. Es decir, cuando las clases son instanciadas, se vuelven objetos.
- Una clase es el plano que permite crear muchos objetos del mismo tipo.

Programación Orientada a Objetos

En OOP la clase es la unidad de programación y se puede ver como un plano del objeto a crear.

El plano o clase, describe como crear algo, sus características y que acciones puede hacer cuando este se crea.



Uno de los principales objetivos en OOP es la reutilización y la mejora en el mantenimiento del código.

Conceptos Fundamentales de Clases

Abstracción: es el acto de identificar las características principales de un objeto para resolver un problema, mientras que se ignoran detalles irrelevantes. Este concepto se usa en OOP para simplificar la interacción con sistemas complejos, permite al usuario enfocarse en lo que el objeto hace y no como lo hace.

Ejemplo: Un avión puede girar a la derecha, pero no es necesario que el usuario conozca lo que ocurre internamente en el avión, sino como lograr el giro.

Encapsulamiento: Esta es la agrupación de acciones y datos que operan en una clase, las cuales cumplen con ciertas restricciones de acceso (ocultación de datos o data hiding). Este concepto protege el estado interno de un objeto de mal o indebido uso del mismo. El manejo de estados internos se realizan con funciones método denominadas “setter” y “getter”.

Ejemplo: El motor de un auto no debería de cambiar su número de serie (no debería existir una función “setter”), pero si se puede leer con un “getter”.

Conceptos Fundamentales de Clases (cont'd)

Herencia: es el mecanismo por el cual una nueva clase (clase derivada o subclase) puede heredar atributos y métodos de una clase ya existente (clase base o superclase). La clase derivada puede extender o modificar el comportamiento de la clase base. De esta forma, se obtiene reusabilidad de código, creándose una jerarquía de clases.

Ejemplo: la clase monoplano puede ser la clase base de la clase biplano, donde se agrega la funcionalidad requerida que contempla el segundo juego de alas.

Polimorfismo: Es la capacidad de las clases para tomar distintas formas mediante el uso de una interfaz común que permite que objetos de diferentes clases sean tratados en forma similar. Esto permite que el código sea más flexible y mantenible.

Ejemplo: el ruido de un motor de auto puede ser distinto dependiendo del modelo, no obstante, todos los motores hacen ruido (son similares).

Implementación de Conceptos de Clases

Las clases implementan estos 4 conceptos mediante la definiciones de sus datos y acciones mediante:

- Atributos: son los estados o las propiedades de un objeto.
- Métodos: son las acciones y los comportamientos del objeto.

Por ejemplo, considere un auto, dependiendo de la aplicación, el mismo puede contener:

- Atributos: modelo, fabricante, año de fabricación, kilómetros recorridos, nivel de aceite, color, cantidad de puertas, combustible o eléctrico, motor apagado?, etc.
- Métodos: conducir recto, mostrar kilómetros recorridos, abrir puertas, obtener nivel de aceite, apagar motor, frenar, etc.

Clases y Objetos

Una clase es el plano de lo que se desea representar, esta incluye los atributos y métodos del objeto.

La clase deben ser escrita, no poseen tiempo de vida porque no ocupan memoria (más allá de la memoria del programa) en tiempo de ejecución.

Al instanciar una clase, se crean objetos. Los objetos tienen un tiempo de vida durante el cual se van modificando o no. Es decir, dos objetos creados de la misma clase, no tienen porque contener la misma información (un auto puede recorrer más camino que otro).

Los objetos hacen el trabajo computacional de nuestros programas.

Ejemplo de una Clase Simple

```
#include <iostream>
#include <string>

class Plane {
public:
    float altitude;    // Altitud del avion
    std::string model; // Modelo de avion
    int capacity;      // Numero de pasajeros

    void mostrarInfo(){
        std::cout << "Modelo: " << model
                    << ", Capacidad: " << capacity
                    << ", Altitude: " << altitude
                    << std::endl;
    }

    void volar() const{
        std::cout << "El avion " << model
                    << " esta volando a una altitud " << altitude
                    << " mts." << std::endl;
    }
};
```

```
int main(){
    Plane myPlane;
    myPlane.model = "Cessna 172 Skyhawk";
    myPlane.capacity = 4;
    myPlane.altitude = 0.0;

    myPlane.mostrarInfo();
    // Imprime:
    //Modelo: Cessna 172 Skyhawk, Capacidad: 4, Altitude: 0
    return 0;
}
```

Programación con Clases

Se puede observar que una clase, dependiendo de la cantidad de métodos y atributos, puede llegar a ocupar una cantidad considerable de líneas y casi nunca en OOP se trabaja con una única clase.

Debido a esto, es conveniente utilizar archivos header y archivos de source para separar la interfaz o definición (que es lo que la clase puede hacer) de la implementación (como hace la clase para realizar las cosas), respectivamente.

Es recomendable tener una clase en cada archivo *header* y uno para su implementación (archivo *source*). No obstante, si las clases están íntimamente relacionadas (ejemplo: Plane y PlaneFuel), puede utilizarse el mismo archivo.

Antes de...

Antes de escribir en forma ordenada la clase conviene comentar dos cosas:

1. En el código de la clase, se observa la palabra public. Esta indica que los métodos y atributos a continuación son de acceso público, se pueden acceder una vez definido el objeto.

Otro calificador que veremos es private. Este indica que los métodos y atributos definidos a continuación, sólo pueden ser accedidos desde dentro del objeto.

Por ejemplo, un atributo private puede ser el estado del aceite, el cual no se desea compartir, pero se utiliza para estimar, con un método Público o privado, si se requiere mantenimiento.

```
class Plane {  
    public:  
        float altitude;           // Altitud del avion  
        std::string model;        // Modelo de avion  
        int capacity;             // Numero de pasajeros  
  
    void mostrarInfo(){  
        std::cout << "Modelo: " << model  
                    << ", Capacidad: " << capacity  
                    << ", Altitude: " << altitude  
                    << std::endl;  
    }  
}
```


Antes de... (cont'd)

2. Durante la compilación, los archivos *header* son compilados como código de objetos cuando el compilador los encuentra.

En caso de cometer el error en que se defina dos veces la misma clase, se genera un error que puede ser muy difícil de resolver, sobre todo en proyectos de gran tamaño. Para evitar esto, hay dos formas posibles de hacerlo:

El caso de “include guards”, es la forma en que se hacía previamente y se sigue soportando por “backward compatibility” (más confiable).

El segundo caso es `#pragma once` que es mas moderno y debido a eso puede tener problemas de compatibilidad.

```
#ifndef PLANE_H  
#define PLANE_H
```

```
class Plane {  
    :  
};
```

```
#endif
```

Include guards

```
#pragma once
```

```
class Plane {  
    :  
};
```

Pragma once

Programación con Clases: header file

En el archivo *header*, no debe estar incluido el archivo *source* “Plane.cpp”.

De hecho, es mala práctica incluir archivos .cpp en un header file, puede producir problemas de múltiples definiciones.

Se agregó el método “requiereMantenimiento”, que devuelve un bool, y el atributo privado “oilStatus”, que mide en porcentaje el nivel de aceite.

Notar que no hay asignación de valores en ningún atributo.

No obstante, atributos del tipo static pueden declararse en el archivo header.

```
#pragma once
```

```
#include <string>
```

```
class Plane {  
public:  
    float altitude;           // Altitud del avion  
    std::string model;        // Modelo de avion  
    int capacity;             // Nr de pasajeros  
  
    void mostrarInfo() const;  
    void volar();  
    bool requiereMantenimiento() const;  
  
private:  
    float oilStatus;          // En porcentaje  
};
```

Programación con Clases: source file

En el archivo de *source* se encuentra el código que implementa las definiciones del archivo *header*.

Se observa que los métodos comienzan con “Plane::”.

La sintaxis “::” se denomina el operador de resolución de scope o “scope resolution operator”.

Así, “Plane::” indica que los métodos pertenecen a la clase “Plane”.

El programa imprimirá:

Modelo: Cessna 172 Skyhawk, Capacidad: 4, Altitude: 0
No volar, requiere mantenimiento,

```
#include "Plane.h"
#include <iostream>

void Plane::mostrarInfo() const{
    std::cout << "Modelo: " << model
                << ", Capacidad: " << capacity
                << ", Altitude: " << altitude
                << std::endl;
}

void Plane::volar(){
    std::cout << "El avion " << model
                << " esta volando a una altitud " << altitude
                << " mts." << std::endl;
}

bool Plane::requiereMantenimiento() const{
    return (oilStatus < 50.0 ? true : false);
} // Si hay menos del 50 % requiere mantenimiento
```

Programación con Clases: main file

Se observa que el archivo “Plane.h” debe incluirse en el main file.

En este programa se agregó un método privado en la clase, que permite determinar si se requiere mantenimiento.

Si se requiere mantenimiento, no se puede volar.

Esta responsabilidad se dejó al piloto, pero no tiene que necesariamente ser así.

```
#include "Plane.h"
#include <iostream>

int main(){
    Plane myPlane;
    myPlane.model = "Cessna 172 Skyhawk";
    myPlane.capacity = 4;
    myPlane.altitude = 0.0;

    myPlane.mostrarInfo();
    // Imprime: Modelo: Cessna 172 Skyhawk, Capacidad: 4, Altitude: 0

    if (myPlane.requiereMantenimiento())
        std::cout << "No volar, requiere mantenimiento" << std::endl;
    else{
        std::cout << "Puede volar, no se requiere mantenimiento." <<
        std::endl;
        myPlane.volar();
    } // No vuela porque oilStatus no se definio

    return 0;
}
```

Miembros Public y Private

Es importante tomarse el tiempo para definir una clase y el acceso que se le otorga a sus métodos y atributos.

No se puede hacer que todos los métodos y atributos sean *public* porque:

- Rompe la regla de encapsulamiento,
- Da a otros objetos acceso total,
- Expone la complejidad de los métodos a los usuarios,
- Otros programadores pueden hacer cambios en partes del sistema que introduzcan bugs.

I AM WATCHING YOU



PYTHON

Como Crear una Clases (primer aprox.)

Para esto es bueno preguntarse:

1. Que hace mi objeto en mi programa? Cuales son sus comportamientos? Estos son usualmente implementados como métodos del tipo público (recuerde encapsulamiento y abstracción)
2. Que estados y datos se necesitan en la clase? Si hay información que sólo es necesaria que la sepa la clase, estos serán datos privados.
3. Recuerde agregar constructores, destructores y operadores sobrecargados (que esto!?)

Constructores

El constructor es la parte del código que se utiliza para inicializar un objeto.

Este es automáticamente ejecutado cada vez que un objeto es creado, permitiendo inicializar los atributos.

Los atributos sólo pueden ser inicializados en el constructor o mediante los métodos setters.

En una clase, siempre se tiene declarado un constructor default (puede no hacer nada) que no es necesario el definirlo.

Un constructor puede ser sobrecargado, es decir, se puede inicializar una clase con distintos tipos de datos.

Constructores - Ejemplo

```
#include <iostream>
#include "Plane.h"

int main(){
    Plane aPlane(10, 20);
    aPlane.setNumWheels(2);

    std::cout << "Numero de ruedas: " <<
    aPlane.getNumWheels() << std::endl;
    // Imprime: Numero de ruedas: 2

    return 0;
}
```

Archivo “main.cpp”

```
#include "Plane.h"
#include <iostream>

Plane::Plane(){
    iNumWheels = 3;
}
Plane::Plane(PLANE_TYPE t){
    if (t==UAV)
        iNumWheels = 0;
    else
        iNumWheels = 3;
}
Plane::Plane(int x, int y){
    iXPos = x; iYPos = y;
}
int Plane::getNumWheels(){
    return iNumWheels;
}
void Plane::setNumWheels(int nrW){
    iNumWheels = nrW;
}
```

Archivo “Plane.cpp”

Constructores – Ejemplo (cont'd)

En el *header*, se pueden observar tres definiciones de constructores.

Estos son ejecutados dependiendo de los tipos de variables utilizados al inicializar el objeto.

Al inicializar un objeto, primeramente se ejecuta la definición de las variables (en este caso, todas son *private*), luego se llama al constructor y se ejecuta su código.

No necesariamente todos los atributos privados deben tener *setters* y *getters*, solo aquellos que se quieren exponer.

```
enum PLANE_TYPE{ JET, PROPELER, UAV, MILITARY};

class Plane {
public:
    Plane();           // Constructor default a definir.
    Plane(PLANE_TYPE t); // Constructor overloaded
    Plane(int x, int y); // Constructor overloaded
    void setNumWheels(int nrW);
    int getNumWheels();

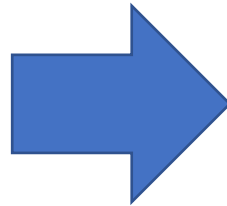
private:
    int iNumWheels;
    PLANE_TYPE planeType;
    int iXPos;
    int iYPos;
};
```

Constructores – Lista de inicialización

Datos miembro pueden ser inicializados con lista de inicialización.

Para el caso de asignar datos default, se puede hacer esto si esta definición se encuentra en el constructor default, haciendo:

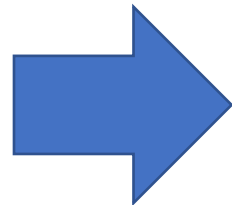
```
Plane::Plane(){  
    iNumWheels = 3;  
}
```



```
Plane::Plane():iNumWheels(3){}
```

Por otro lado, también se puede usar para inicializar variables con valores al momento de inicializar el objeto:

```
Plane::Plane(int x, int y){  
    iXPos = x; iYPos = y;  
}
```



```
Plane::Plane(int x, int y):iXPos(x), iYPos(y){}
```

Las variables tipo **const** deben ser inicializadas utilizando la lista de inicialización.

Constructores – Lista de inicialización (cont'd)

Las variables tipo **const** deben ser inicializadas utilizando la lista de inicialización.

En el archivo header:

```
class Plane {  
    public:  
        Plane();  
        Plane(PLANE_TYPE t);  
        Plane(int x, int y);  
        void setNumWheels(int nrW);  
        int getNumWheels();  
  
    private:  
        int iNumWheels;  
        PLANE_TYPE planeType;  
        const int iXPos;  
        int iYPos;  
};
```

En el archivo source sin lista de inicialización:

```
Plane::Plane(int x, int y){  
    iXPos = x; // Error de compilación porque iXPost es const  
    iYPos = y;  
}
```

En el archivo source con lista de inicialización:

```
Plane::Plane(int x, int y):iXPos(x), iYPos(y){}  
// Sin errores de compilación
```

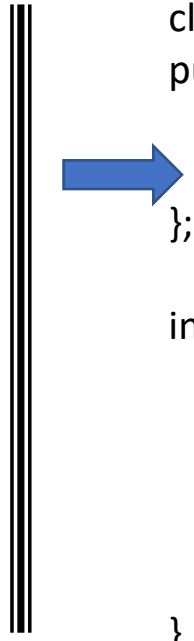
Constructores – Llamada Implícita / Explícita

Un constructor puede invocarse con una llamada implícita o explícita.

Una invocación implícita la hace el compilador y es automática, puede conllevar una conversión de tipo, no utiliza paréntesis e implica menor control.

Una invocación explícita se hace directamente con la instrucción del programador, utiliza paréntesis y se tiene control de la llamada.

```
class myClass {  
public:  
    int val;  
    myClass(int i): val(i) {}  
};  
  
int main(){  
    myClass obj = 2; // Invocacion Ilmplicita, val = 2  
    obj = 20;        // Implicita, val = 20 (temp obj y copia)  
    myClass obj2(4); // Invocacion explicita  
  
    return 0;  
}
```



```
class myClass {  
public:  
    int val;  
    explicit myClass(int i): val(i) {}  
};  
  
int main(){  
    //myClass obj = 2; // Error: No existe constructor  
    myClass obj2(4);   // Invocacion explicita  
    //myClass obj2(5); // Error por redeclaracion  
  
    return 0;  
}
```


Constructores – Llamada Implícita No Deseada

En el código de la derecha, hay una función que recibe un objeto como parámetro.

En caso de pasar un *int* a la función, esto permitirá crear un objeto llamando en forma implícita a su constructor.

Esto hará que se cree un objeto que existirá en el scope de la función print.

Genera una potencial fuente de bugs y funcionalidad no deseada.

Para evitar esto, es necesario declarar al constructor como explicit, lo que producirá que no se pueda compilar el código.

```
#include <iostream>

class myClass {
public:
    myClass(int x) { std::cout << "Se llamo al constructor con: " << x << "\n"; }
};

// Al llamar a la función se llama implícitamente al constructor con 1
void print(myClass obj) { std::cout << "Se llamo a print" << std::endl; }

int main() {
    print(1); // Se llama a print con un número
}
```

Destructores

- Al igual que un constructor inicializa un objeto, un destructor permite liberar memoria que haya sido dinámicamente requerida durante la vida del objeto.
- El destructor será ejecutado cuando el objeto salga de scope, por ejemplo, termina una función o el programa finaliza.
- Definir el destructor es necesario sólo cuando se utilizan punteros que deben ser liberados.
- En el código, se observa la definición del destructor “~Plane()”
- Existe un atributo privado que es un puntero a un objeto de la clase Engine.
- El destructor debe liberar la memoria del puntero Engine.

Declaración
del destructor

```
#include <iostream>

class Plane {
public:
    Plane(int x, int y);
    ~Plane();

private:
    Engine* planeEngine;
};
```

Destructores (cont'd)

- De no escribir un destructor, cuando el objeto sale de scope, el puntero *planeEngine* se pierde, causando un memory leak.
- Usualmente en el destructor se hace un free/delete de los miembros datos que son punteros. En el caso de ser necesario, se les asigna un *nullptr* para evitar dangling pointers.
- Note que en este caso sólo se puede tener un destructor por clase.

```
Plane::~~Plane(){  
    delete planeEngine;  
    planeEngine = nullptr; // Asigna nullptr al puntero  
}
```

Objetos Creados Dinámicamente

- Los objetos puede crearse dinámicamente como el resto de las variables.
- Esto puede llevarse a cabo con Smart Pointers o con el comando *new*.
- Cuando el objeto es un puntero, se usa el operador “->” para llamar los métodos o las variables públicas.
- El operador *new* se encarga de:
 - Alocar memoria para el objeto,
 - Llamar al constructor del objeto,
 - Devolver un puntero al objeto creado.
- El operador *delete* elimina los datos de la memoria *heap* para luego redirigir el puntero a *nullptr* (dirección 0x0).

```
#include <iostream>
#include "newPlane.h"

int main(){
    Plane* ptrToPlane = new Plane();
    ptrToPlane->setNumWheels(20);

    delete ptrToPlane;
    ptrToPlane = nullptr;

    return 0;
}
```

El Puntero *this*

- El puntero *this* hace referencia a la instancia actual de la clase (el objeto).
- Así, *this* se puede utilizar cuando se necesita hacer referencia, como pueden ser los casos en que se quiera:
 - Resolver una ambigüedad entre los nombres de los atributos y parámetros pasados a un método.
 - Devolver el objeto, por ejemplo, para encadenar métodos para agilizar la programación.
 - Pasar el objeto a una función.
 - Etc.

El Puntero *this* (cont'd)

```
#include <iostream>
```

```
class MyClass {
```

```
private:
```

```
    int x;
```

```
public:
```

```
    MyClass(int val) : x(val) {}
```

```
    void setX(int x) {
```

```
        this->x = x; // Usando 'this' por la ambigüedad con x
```

```
    }
```

```
    MyClass* multiplyX(int factor) {
```

```
        this->x *= factor;
```

```
        return this; // Devolviendo 'this' para poder encadenar
```

```
    }
```

```
    void display() const {
```

```
        std::cout << "x = " << x << std::endl;
```

```
    }
```

```
};
```

```
int main() {
```

```
    MyClass obj(10);
```

```
    obj.display(); // Imprime: 10
```

```
    obj.setX(20);
```

```
    obj.display(); // Imprime: 20
```

```
    // Metodos encadenando usando 'this'
```

```
    obj.multiplyX(2)->display();
```

```
    // Imprime: 40
```

```
    return 0;
```

```
}
```


Funciones Lambda con Objetos - Ejemplo

Además de las opciones de captura de variables ya vistas para funciones lambda, también se puede usar *this*:

- [=] - todas las variables en el scope de la función lambda (excepto las variables static, extern o thread_local) son capturadas por valor.
- [&] - todas las variables en el scope de la función lambda (excepto las variables static, extern o thread_local) son capturadas por referencia.
- [this] - “this” es el puntero que apunta al objeto actual dentro de un miembro de clase capturado por referencia.

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

class MyClass {
public:
    int value;
    MyClass(int val) : value(val) {}

    void display() { // Funcion lambda que captura el objeto por referencia
        auto lambda = [this]() {
            cout << "Value: " << value << endl; // Accediendo el miembro 'value'
        };
        lambda(); // Llama a la función lambda
    }
};

int main() {
    MyClass obj(10);
    obj.display(); // Imprime: "Value: 10"
    return 0;
}
```

Preguntas?