

Vigésimo primera Sesión

Metodologías y Técnicas de Programación II

**Programación Orientada a
Objeto (POO)
C++**

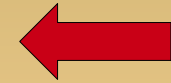
Polimorfismo II

Estado del Programa

Introducción a la POO

Historia de la Programación
Conceptos de POO

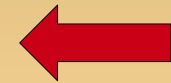
C++
Mi primera Clase



Repaso de Conceptos

Estándares de Programación
Punteros y Memoria

E/S
Control y Operadores



Clases y Objetos en C++

Uso y aplicación
Constructores
Constantes e "inline"

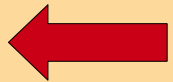
Funciones Amigas
Sobrecarga de Funciones



Sobrecarga

De Operadores

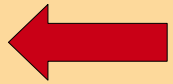
Creación Dinámica de Objetos



Herencia.

Tipos de Visibilidad

Herencia Múltiple



Polimorfismo

Funciones Virtuales

Polimorfismo y Sobrecarga.



Plantillas

Contenedores

Iteradores

21.1 Repaso

Concepto de Superposición

```
class ClaseA
{
    public:
        void Incrementar()
        {
            cout << "Suma 1" << endl;
        }
        void Incrementar(int n)
        {
            cout << "Suma " << n << endl;
        }
};
```

```
Suma 2
Suma 1
Suma 10
```

```
class ClaseB : public ClaseA
{
    public:
        void Incrementar()
        {
            cout << "Suma 2" << endl;
        }
};
int main()
{
    ClaseB objeto;
    objeto.Incrementar();
    //objeto.Incrementar(10); ¡NO EXISTE!
    objeto.ClaseA::Incrementar();
    objeto.ClaseA::Incrementar(10);
    cin.get();
    return 0;
}
```

21.1 Repaso

Polimorfismo

Es la tercera característica que hemos visto de la POO:

- Abstracción de Datos
- Herencia
- Polimorfismo

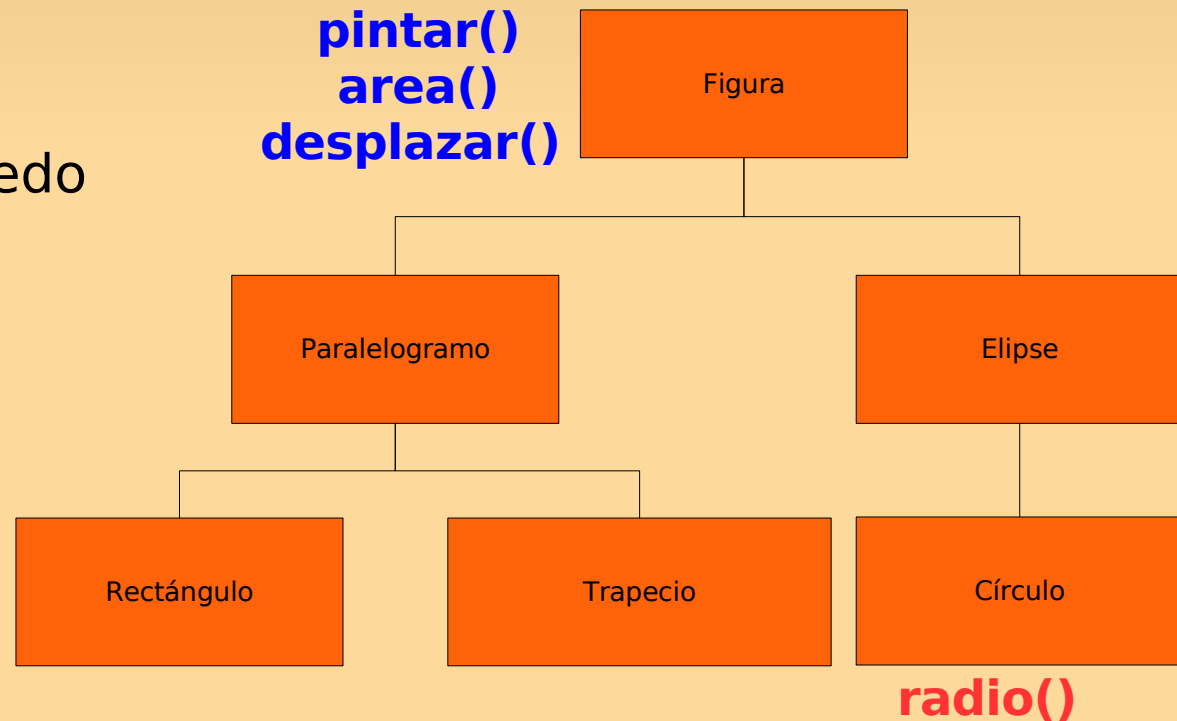
Si p es un puntero a “figura” puedo moverme por la jerarquía.

No podemos hacer esto:

```
p->radio();
```

Pero sí esto:

```
p->pintar();
```



Atención que el polimorfismo aplica en situaciones en las que usamos punteros o referencias.

21.1 Repaso

Definición más clásica de Polimorfismo

Una variable pasada o esperada puede adoptar múltiples formas.

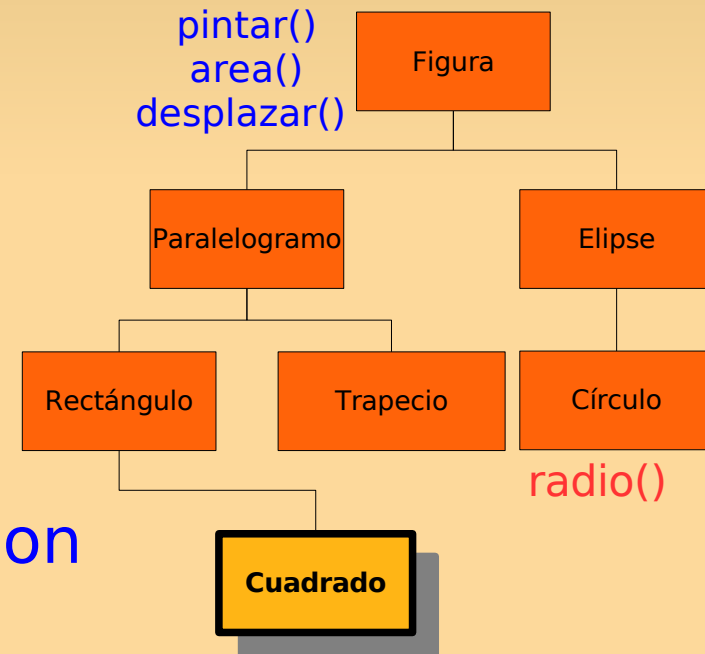
Cuando se habla de polimorfismo en POO se entienden dos cosas:

1.- Se puede trabajar con un objeto de una clase sin importar de qué clase se trata.

2.- Posibilidad de declarar métodos con el mismo nombre que pueden tener diferentes argumentos dentro de una misma clase.

Para nosotros 2 es : _____.

La capacidad de un programa de trabajar con más de un tipo de objeto se conoce con el nombre de polimorfismo.



21.1 Repaso

Comportamiento por defecto

Cuando se invoca un método de un objeto de la clase derivada mediante un puntero a un objeto de la clase base, se trata al objeto como si fuera de la clase base.

Hasta ahora la herencia se ha utilizado:

Para heredar los miembros de una clase base.

Existe la posibilidad de que un método de una clase derivada se llame como método de la clase base pero tenga un funcionamiento diferente.

¿Os acordáis del problema con la superposición y la vinculación?

21.1 Repaso

Comportamiento por defecto:

```
class Instrumento
{
    public:
        void tocar(nota n) const;
};

class Instrumento_Viento : public
Instrumento
{
    public:
        void tocar(nota n) const ;
};

void afinar(Instrumento& i)
{
    // ...
    i.tocar(Do);
}

Instrumento_Viento flauta;
afinar(flauta);
```

La función afinar() acepta por referencia un instrumento.

Por defecto se llama a tocar de la clase base.

Si declaramos tocar() como virtual en la clase base Instrumento cambiamos el comportamiento por defecto.

“Enviamos un mensaje al objeto y dejamos que sea el objeto el que se preocupe de qué hacer con él”

21.1 Repaso

Conversión entre Objetos

Un objeto de la clase base también “ES UN” objeto de la clase derivada.

```
Alumno a;  
Persona p;  
.....  
p = a;  
a = p; // No!!
```

Solo convertimos hacia el sentido en que se cumplen interfaces.

Vincular: Relacionar la **llamada** a una función con el cuerpo o la **implementación** de la función.

De forma estática: Antes de ejecutar.

De forma dinámica: En tiempo de ejecución según la clase del objeto.

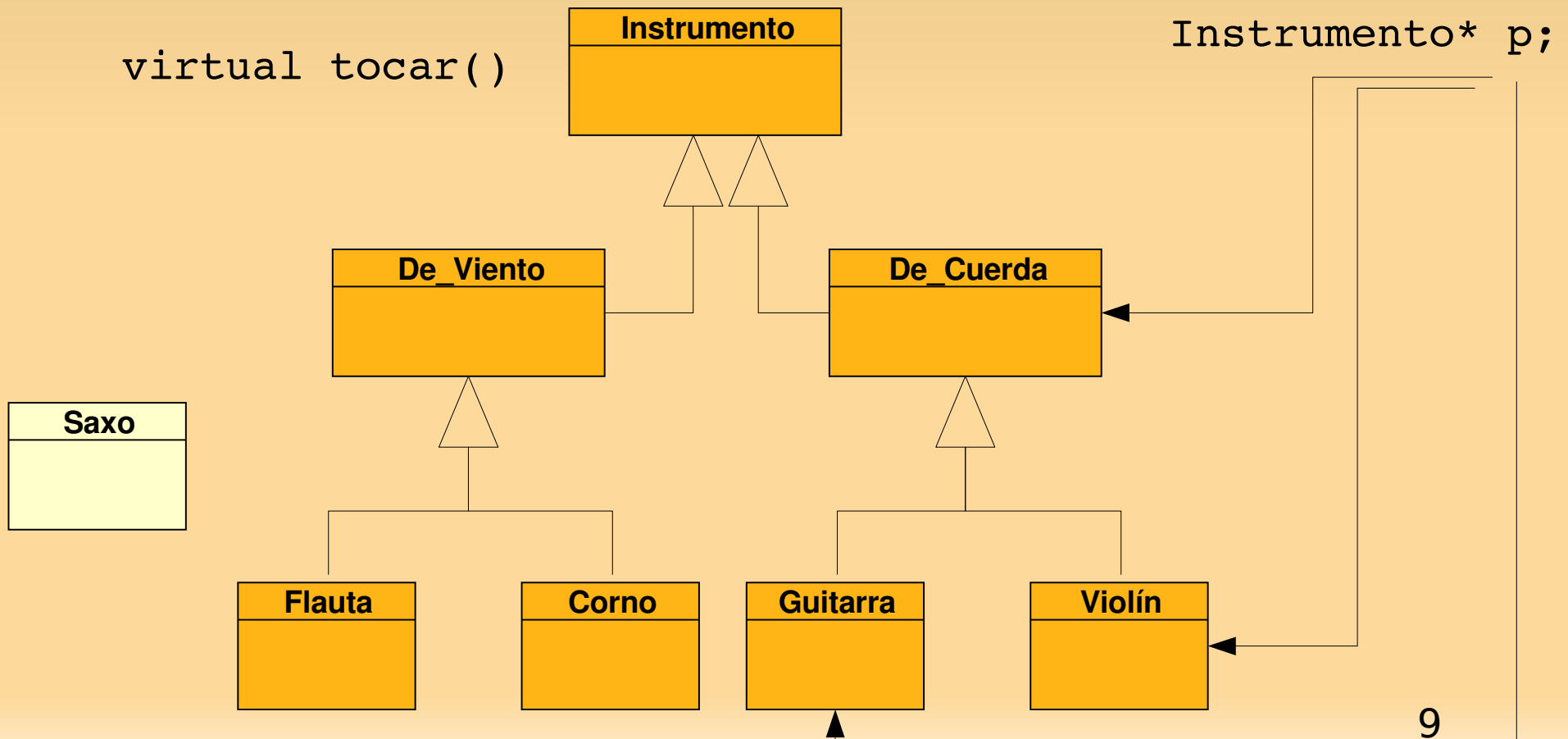
Para decirle al compilador que haga la vinculación en tiempo de ejecución basándose en el tipo o la clase del objeto utilizamos:

virtual

21.1 Repaso

Un punteros, una jerarquía de clases y funciones virtuales....

Nos permite hacer una función genérica `afinar()`.
Si otro programador añade un instrumento mi programa:
¡sigue funcionando!



21.2 Polimorfismo

Clases Abstractas

A menudo en el diseño lo que queremos es utilizar la clase base para presentar un interfaz para sus clases derivadas.

No me interesa que haya objetos de esa clase:

- Sólo quiero que se deriven otras clase de ella.
- Las clases derivadas tienen las funciones de la base por herencia.
- La nueva clase sirve para hacer “upcasting” (conversión hacia arriba).

Esto se consigue creando una **clase abstracta** poniendo como mínimo una **función virtual pura**.

- Usa la palabra reservada “virtual”.
- Es seguida por “=0”.

21.2 Polimorfismo

Clases Abstractas

Las funciones virtuales puras me permiten poner una función en el interfaz sin tener que poner un código que no interesa.

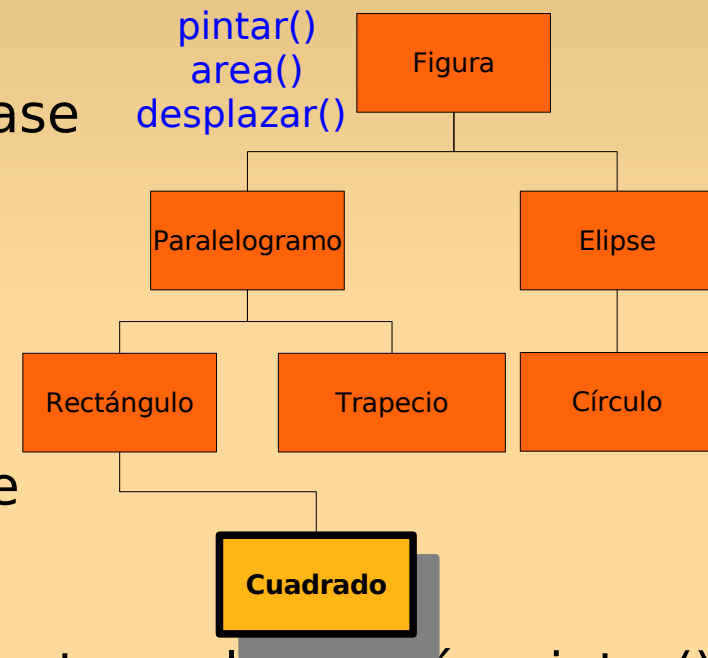
¿Para qué poner código dentro de `pintar()` de la clase `Figura`?

Lo que interesa es que haya código para pintar en las clases derivadas. Cuadrado, Círculo,...

Además obligamos a que las clase derivadas a que implementen código para las funciones virtuales.

En el ejemplo, y hasta ahora, poníamos código tonto en la función `pintar()` como mostrar algo por pantalla.

Que se llame a la función `Figura::pintar()` es que algo anda mal. No tiene sentido.



21.2 Polimorfismo

Clases Abstractas

```
class Figura
{
    public:
        void pintar() const = 0;
};
class Elipse : public Figura
{
    public:
        void pintar() const {
            cout << "Elipse::pintar()"
        };
};
Figura& rfig;
Figura* pfig;
Elipse  mi_elipse;
rfig = mielipse;
pfig = &mi_elipse;
rfig.pintar();
pfig->pintar();
```

La función **pintar()** es una **función virtual pura**.

La clase Figura es una clase abstracta.

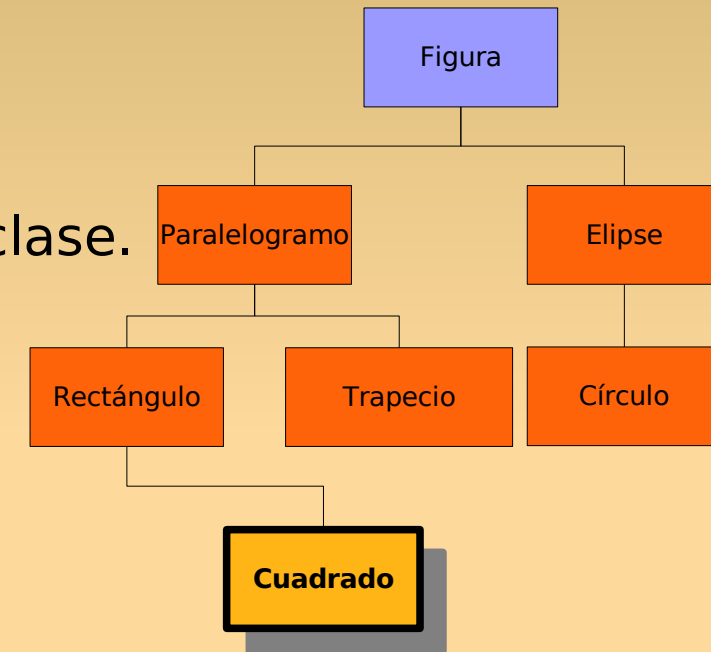
21.2 Polimorfismo

Clases Abstractas y Funciones Virtuales Puras

Las funciones virtuales son útiles porque:

Hacen explícita la abstracción de una clase.

Indican al usuario y al compilador cómo se usa la clase.



Una clase abstracta no se puede pasar por valor.

Al convertir una clase en abstracta también garantizamos que siempre se use un puntero o una referencia cuando se haga “upcasting” a esa clase.

Podemos tener funciones virtuales puras y funciones “normales” en la misma clase A.

No puedo crear objetos de A pero tengo el código común lo más “arriba” posible.

21.2 Polimorfismo

Clases Abstractas

```
class Mascota {
public:
    virtual void comer() const = 0;
    //! virtual void dormir() const = 0{}
};

void Mascota::comer() const {
    cout <<"Mascota::comer()"<< endl;
}

class Perro : public Mascota {
public:
    void comer() const {
        Mascota::comer();
    }
};

int main() {
    Perro simba;
    simba.comer();
}
```

La función **pintar()** es una **función virtual pura**.

Podemos poner código en una función virtual pura.

No podemos hacerlo de forma "inline" en la declaración (Ver dormir()).

Es una forma de poner código que luego sea reutilizable.

21.2 Polimorfismo

Destructores Virtuales

Supongamos que tenemos una estructura de clases en la que en alguna de las clases derivadas exista un destructor.

Un destructor es una función como las demás, por lo tanto, si destruimos un objeto referenciado mediante un puntero a la clase base, y el destructor no es virtual, estaremos llamando al destructor de la clase base.

Esto puede ser desastroso, ya que nuestra clase derivada puede tener más tareas que realizar en su destructor que la clase base de la que procede.

Por lo tanto debemos respetar siempre ésta regla:

“si en una clase existen funciones virtuales, si creamos un destructor éste debe ser virtual”

21.2 Polimorfismo

Destructores Virtuales

Supongamos que tenemos una estructura de clases en la que en alguna de las clases derivadas exista un destructor.

Un destructor es una función como las demás, por lo tanto, si destruimos un objeto referenciado mediante un puntero a la clase base, y el destructor no es virtual, estaremos llamando al destructor de la clase base.

Esto puede ser desastroso, ya que nuestra clase derivada puede tener más tareas que realizar en su destructor que la clase base de la que procede.

Por lo tanto debemos respetar siempre ésta regla:

“si en una clase existen funciones virtuales, si creamos un destructor éste debe ser virtual”

21.2 Polimorfismo

Polimorfismo y Clases Abstractas

Resumiendo mucho:

Dada una clase abstracta, no se pueden crear objetos de esa clase base. Se pueden crear punteros que a objetos de la clase base abstracta que realmente apunten a objetos de la clase derivada.

Polimorfismo significa "formas diferentes".

- Se tiene la misma vista: la interfaz común en la clase base.
- Diferentes formas de usarla: las diferentes versiones de las funciones virtuales.

21.3 Ejercicios

1.- Crea una jerarquía simple "figura": una clase base llamada Figura y una clases derivadas llamadas Circulo, Cuadrado, y Triangulo.

En la clase base, hay que hacer una funcion virtual llamada dibujar(), y sobreescribirla en las clases derivadas.

Hacer un array de punteros a objetos Figura creados con new (en el montón - heap) y que obligue a realizar upcasting de los punteros y llamar a dibujar() a través de la clase base para verificar el comportamiento de las funciones virtuales.

Ver el programa paso a paso usando el depurador.

2.- Modifica el Ejercicio 1 de tal forma que dibujar() sea una función virtual pura. Intenta crear un objeto de tipo Figura. Intenta llamar a la función virtual pura dentro del constructor y mira lo que ocurre. Dejándolo como una funcion virtual pura crea una definicion para dibujar().

3.- Aumentando el Ejercicio 2, crea una funcion que use un objeto Figura por valor e intente hacer un upcast de un objeto derivado como argumento. Vea lo que ocurre. Arregla la función usando una referencia a un objeto Figura.