Metodologías y Técnicas de Programación II

Programación Orientada a Objeto (POO) C++

Polimorfismo I

Estado del Programa

Introducción a la POO
- Historia de la Programación
- Conceptos de POO

Repaso de Conceptos
- Estándares de Programación
- Punteros y Memoria

Clases y Objetos en C++
- Uso y aplicación
- Constructores
- Constantes e "inline"

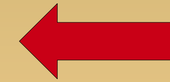
Sobrecarga
- De Operadores

Herencia.
- Tipos de Visibilidad

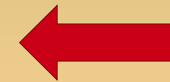
Polimorfismo
- Funciones Virtuales

Plantillas
- Contenedores

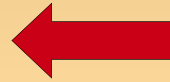
C++
Mi primera Clase



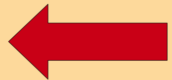
E/S
Control y Operadores



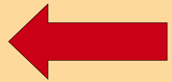
Funciones Amigas
Sobrecarga de Funciones



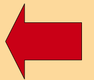
Creación Dinámica de Objetos



Herencia Múltiple



Polimorfismo y Sobrecarga.



Iteradores

20.1 Repaso

Herencia y Composición

Formas de reutilización sistemática. Mejor. Más concienzudas.

Composición: Creamos objetos de la clase existente dentro de la nueva clase que estamos creando.

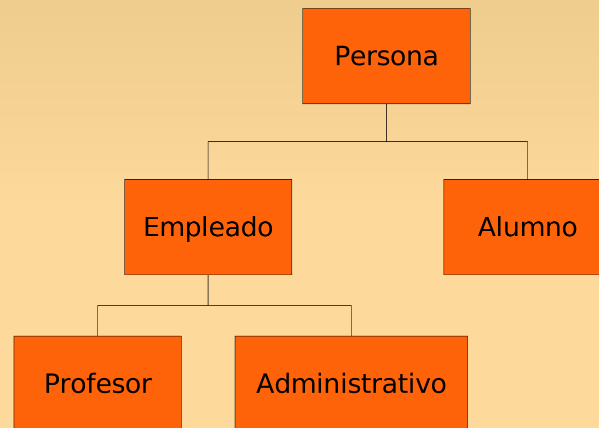
Herencia: Se toma la forma de la clase existente y se añade código sin modificar la clase existente.

20.1 Repaso

Jerarquías de Clases

Cuando una clase deriva de más de una clase base: **Herencia Múltiple.**

Jerarquías de Clases.




```
class <clase_derivada> :
        [public|private] <base1> [,[public|private] <base2>] {};
```

Clase base, clase padre, clase madre, superclase: La clase base es la clase ya creada, de la que se hereda.

Clase derivada, clase hija, clase heredada: es la clase que se crea a partir de la clase base.

20.1 Repaso

Desarrollo Incremental

Es una ventaja de la Herencia y de la Composición.

- Podemos añadir código sin causar fallos en el código existente.
- Si hay fallos se rectifican en el nuevo código.
- Las clases – y el código - quedan separadas de forma limpia.
- Incluso sin tener el código fuente (Cabecera.h y código objeto).
- Permite crecer de forma evolutiva (me baso en lo anterior).

20.1 Repaso

Superposición (Redefinimos en la derivada)

```
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); ¿?
   objeto.ClaseA::Incrementar();
   objeto.ClaseA::Incrementar(10);
   cin.get();
   return 0;
}
```

20.2 Polimorfismo

Polimorfismo

El Polimorfismo (implementado en C++ con funciones virtuales) es la tercera característica esencial de un lenguaje orientado a objetos.

- Abstracción de Datos
- Herencia
- Polimorfismo

Es otra forma de separar interfaz de implementación.

La encapsulación nos permite crear nuevos tipos de datos combinando características (atributos) y comportamientos (métodos).

Control de acceso: Separamos interfaz de implementación haciendo **privados** los detalles.

La herencia nos permite tratar a un objeto como de su propia clase o como su clase base.

20.2 Polimorfismo

Concepto de Polimorfismo

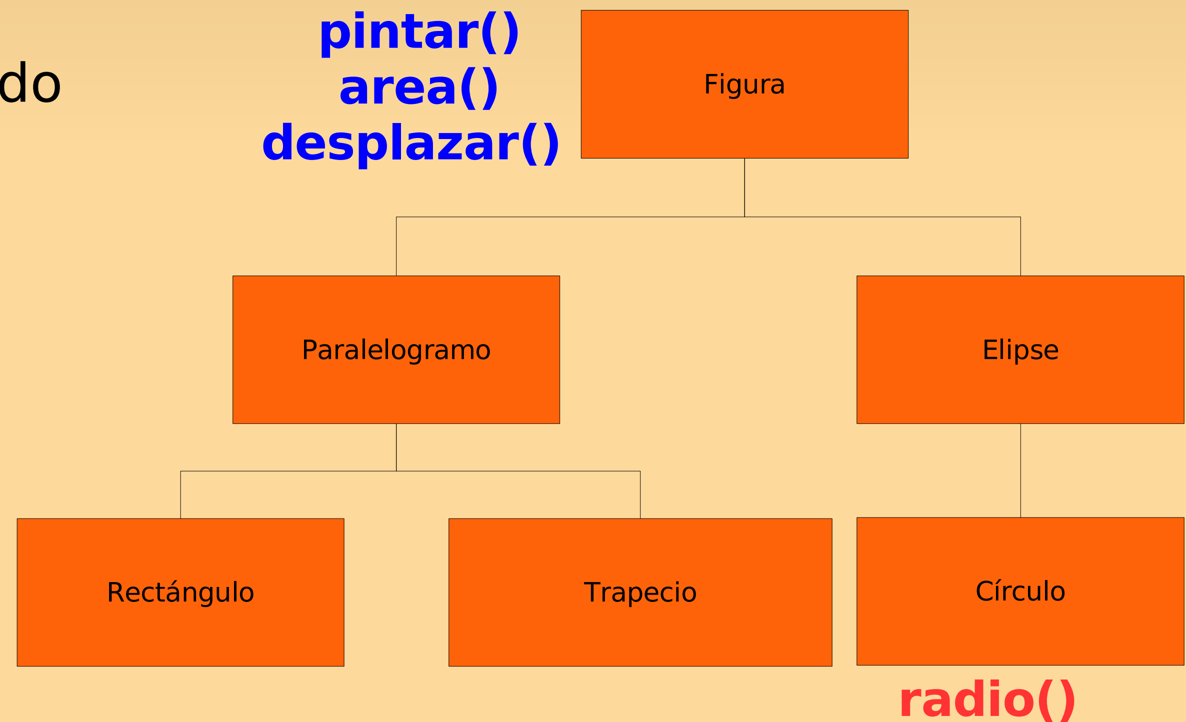
C++ nos permite acceder a objetos de una clase derivada usando punteros a la clase base.

Sólo podemos acceder a atributos y métodos que existan en la clase base:

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

No puedo llamar a radio()

¿Por qué?



20.2 Polimorfismo

Definición más clásica de Polimorfismo

Una variable pasada o esperada puede adoptar múltiples formas.

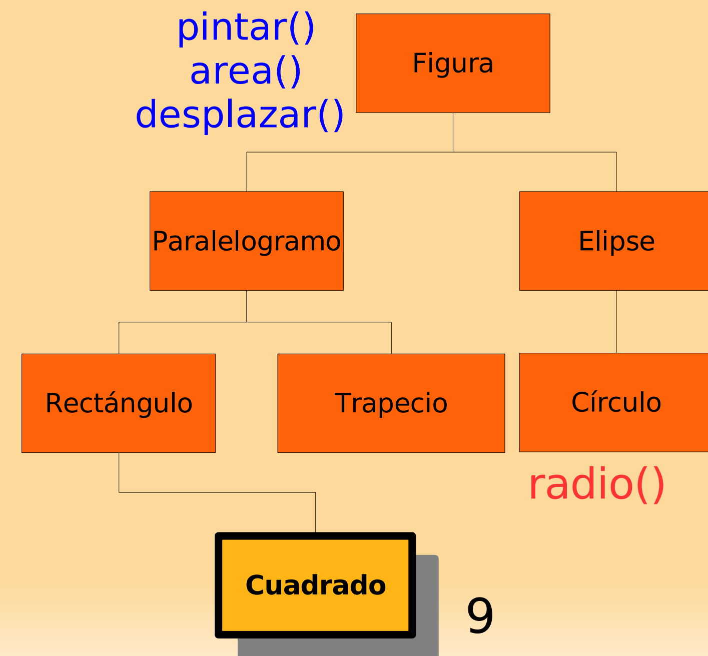
Cuando se habla de polimorfismo en POO se entienden dos cosas:

1.- Se puede trabajar con un objeto de una clase sin importar de qué clase se trata.
 Se trabajará igual sea cual sea la clase a la que pertenece el objeto.
 Esto se consigue mediante jerarquías de clases y clases abstractas.

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

Nosotros a 2 lo hemos llamado: ¿?

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



20.2 Polimorfismo

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?

20.2 Polimorfismo

Problemas con la Vinculación

```
void Persona :: felizcumple()
{
  edad++;
  cout << "Felicidades!!";
  mostrar();
}

int main()
{
  Alumno alum ("77777R", 20,
               "Ana" , "Ruiz", 3 );
  alum.felizcumple();
}
```

El método felizcumple() no está definido en Alumno.

Se busca en persona, y se encuentra.

El mensaje mostrar() se vincula con la clase Persona en lugar de con la clase Alumno.

La salida es:

```
Felicidades!!
77777R Nombre: Ana
```

Nos falta el curso.
Está llamando a mostrar() de persona

20.2 Polimorfismo

Otro ejemplo del problema

```
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.

Terminar de realizar el programa, compilarlo y mostrar su salida por pantlla.

Discusión de problema.

Solución del Problema.

20.2 Polimorfismo

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

20.2 Polimorfismo

Funciones virtuales

```
class Persona
{
    ...
    virtual mostrar();
    ...
};
void Persona :: felizcumple()
{
  edad++;
  cout << “Felicidades!!”;
  mostrar();
}

int main()
{
  Alumno alum (“77777R”, 20,
               “Ana” , “Ruiz”, 3 );
  alum.felizcumple();
}
```

Sólo hay que predecer la declaración de la función con la palabra reservada **virtual**.

Sólo la declaración, no la definición o implementación.

Sólo es necesario declarar la función virtual en la clase base. No es necesario hacerlo en las derivadas.

Todas las funciones de las clases derivadas que encajen con la declaración serán llamadas utilizando el mecanismo virtual.

20.2 Polimorfismo

Funciones Virtuales

Con tocar() definido en la clase base como ***virtual*** podemos añadir tantas nuevas clases como queramos sin que tengamos que tocar una función general como afinar() por el hecho de haber añadido un nuevo instrumento.

1.- Estoy haciendo muy flexible mi código.

2.- Puedo añadir nuevo código en forma de nuevas clases sin “molestar” a programadores que estén usando mis clases base.

3.- Separo el código de unas clases derivadas de otras.

```
virtual <tipo> <nombre_función>(<lista_parámetros>) [{}];
```

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

Saber qué métodos son virtuales es tarea de la fase de diseño. No es fácil.

20.2 Polimorfismo

Funciones Virtuales

Los métodos virtuales heredados son también virtuales en la clase derivada, por lo que si alguna clase hereda de la clase derivada el comportamiento será similar al indicado.

Los únicos métodos que no pueden ser declarados como virtuales son:

- Los constructores
- Los métodos estáticos
- Los operadores new y delete.

Atención, pregunta:

¿Por qué no declaramos siempre los métodos como virtuales y nos quitamos problemas?

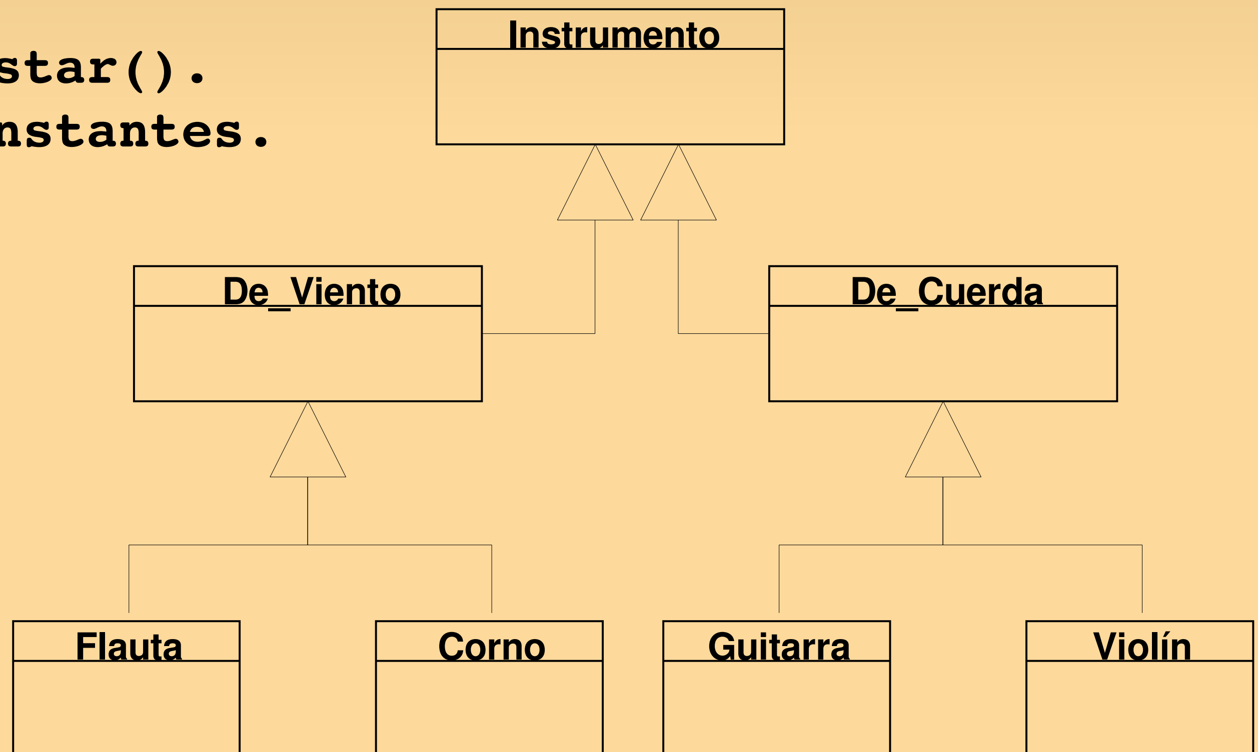
Es una cuestión de rendimiento. La vinculación estática se hace al compilar. La vinculación dinámica tiene que "buscar" durante la ejecución del programa cuál es el código a ejecutar.

20.2 Polimorfismo

Ejercicio

1.- Implementar esta jerarquía de clases con sus métodos sabiendo que todas las clases deben contar con un destructor, un constructor por defecto y las función virtuales tocar(), quien() y ajustar().

Violín no implementa ajustar().
tocar() y quien() son constantes.



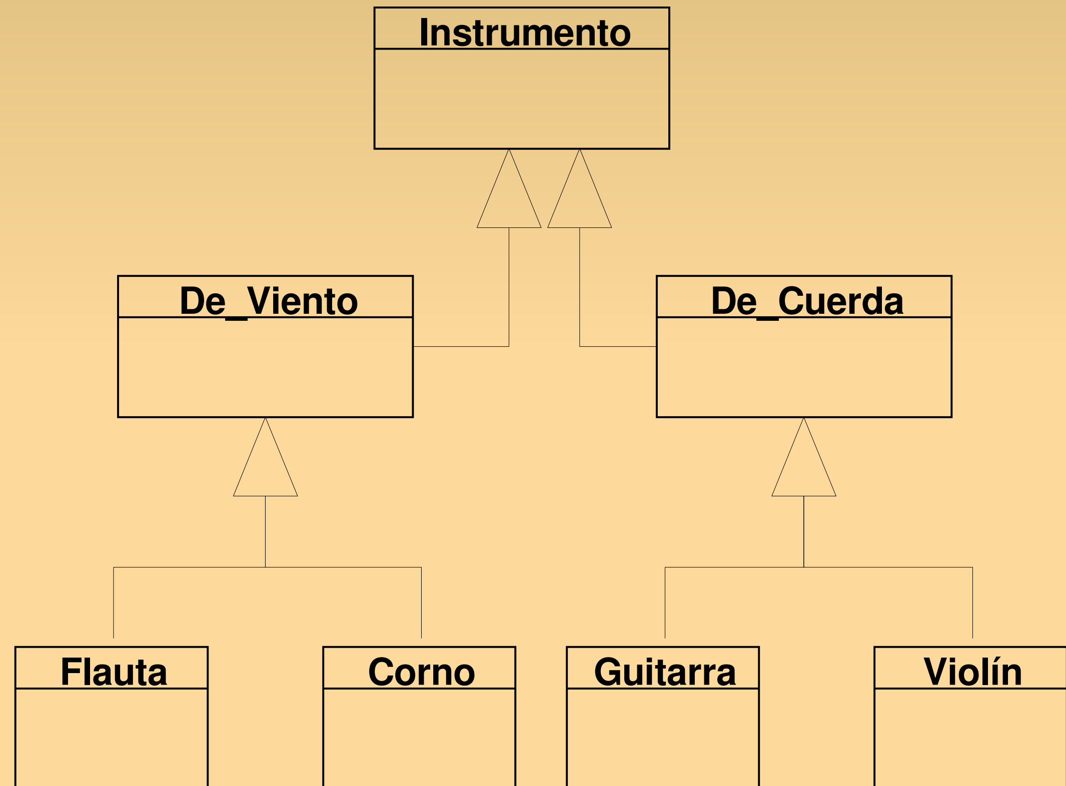
20.2 Polimorfismo

Ejercicio

2.- Mostrar la salida por pantalla de:

```
Instrumento* orquesta[6];
orquesta[0] = new Flauta;
orquesta[1] = new Flauta;
orquesta[2] = new Corno;
orquesta[3] = new Guitarra;
orquesta[4] = new Violin;
orquesta[5] = new Violin;
for (int i=0;i<6;i++)
{
  orquesta[i].quien();
  orquesta[i].ajustar();
}

for (int i=0;i<6;i++)
{
  orquesta[i].quien();
  orquesta[i].ajustar();
}
for (int i=0;i<6;i++)
{
  delete orquesta[i];
}
```



(*) Usar ejercicio preparado....