

PROGRAMACIÓN ORIENTADA A OBJETOS (parte 3)

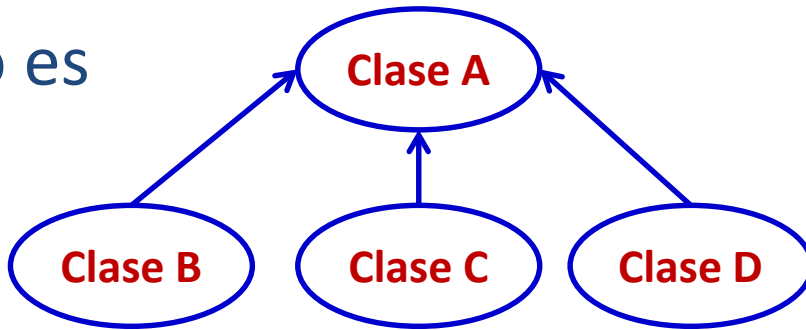
Elaborado por: Juan Miguel Guanira Erazo

PUCP

PUNTEROS EN HERENCIA

Una de las características de la herencia dice que un puntero de clase base puede apuntar directamente a cualquier puntero de clase derivada, sin necesidad de hacer una operación “cast”.

Esto es



```
class ClaseA *pt;  
class ClaseB objB;  
class ClaseC objC;  
class ClaseD objD;
```



```
pt = &objB; ✓ pt = &objC; ✓ pt = &objD; ✓
```

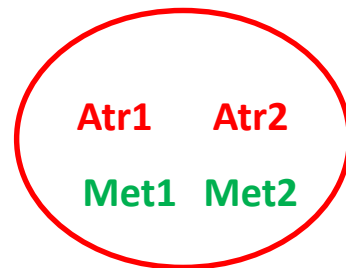
Restricciones:

El puntero de clase base solo podrá acceder a los elementos de la clase base definidos en la clase derivada.

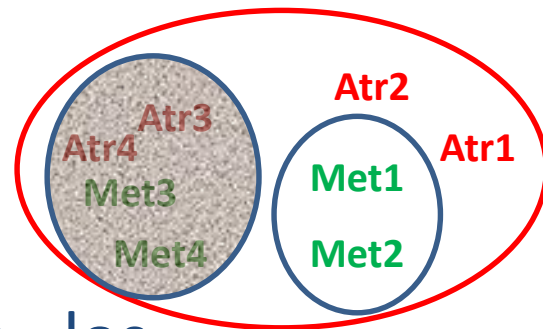
class ClaseBase *pt;



Clases Base



Objeto de Clases Derivada



El puntero no podrá acceder a los elementos propios de la clase derivada.



Pasemos al programa

MÉTODOS VIRTUALES

Todo identificador que define a un método o a una función recibe en tiempo de compilación una dirección fija en el segmento de código.

Cuando se llama a una función en un programa, automáticamente el índice del programa se sitúa en esa dirección y empieza a ejecutar las instrucciones de la función.

class A objA



class B objB



class C objC

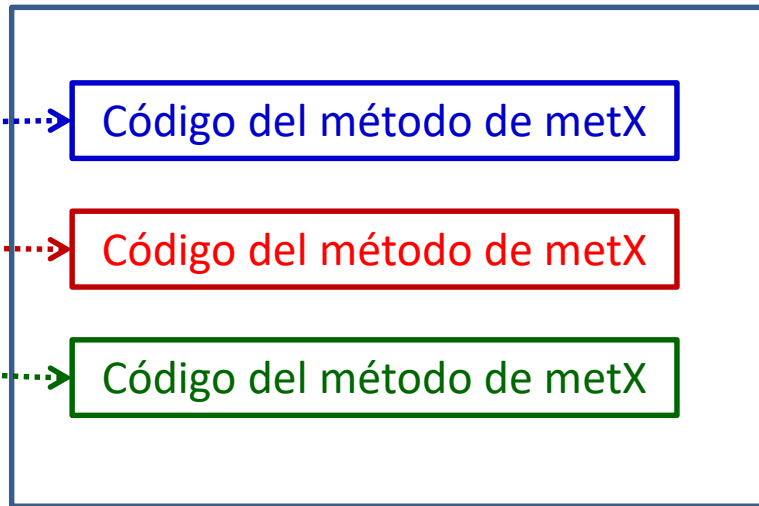


Tabla de identificadores

| | |
|-----------|-----------------|
| objA.metX | DM ₁ |
| objB.metX | DM ₂ |
| objC.metX | DM ₃ |

objC.metX

Segmento de Código



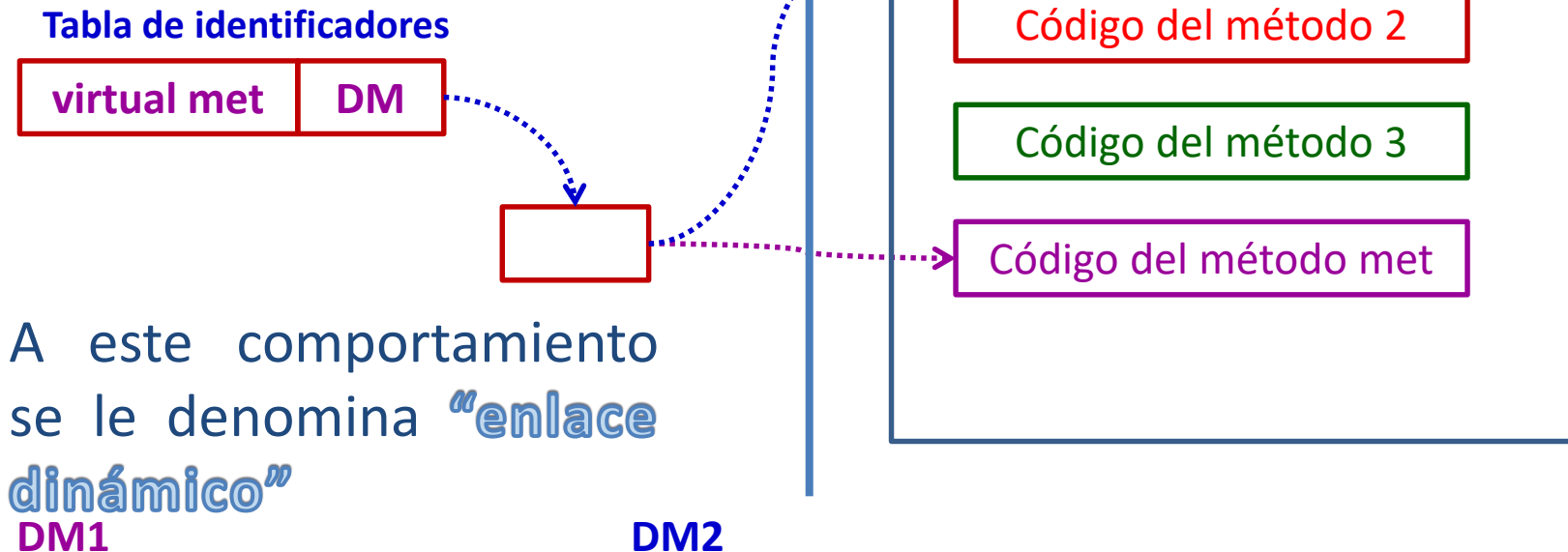
Un método virtual es manejado de manera diferente por el compilador.

Al igual que cualquier método, el compilador le asigna una dirección de memoria, pero ésta no corresponde con el código de inicio de la función, sino que lo hace trabajar como un doble puntero.

Esto quiere decir que en esa posición de memoria se colocará otra dirección de memoria que corresponderá al del código de un método, que no necesariamente será el del método original.

Esta asignación se realizará en tiempo de ejecución.

Comportamiento de un método virtual:



A este comportamiento se le denomina “enlace dinámico”

IMPLEMENTACIÓN DE UN MÉTODO VIRTUAL

La implementación de un método virtual se realiza dentro de una jerarquía de clases (herencia), por lo general en la clase base.

Esta implementación solo tendrá un efecto o significado si el método virtual ha sido sobrescrito en alguna de la clase derivadas y la ejecución se realiza a través de un puntero de la clase base.

Implementación:

```
class Base{  
    private:  
        ...  
  
    public:  
        virtual T metodoA(...);  
        ...  
}
```

```
class Derivada:  
    public Base{  
        private:  
            ...  
        public:  
            T metodoA(...);  
    }
```

Definición:

Si se cumplen todas estas condiciones:

- 1.- Si en un proyecto se implementa una jerarquía de clases.
- 2.- Si en las clases derivadas existen métodos que sobrescriban algunos métodos de la clase base.
- 3.- Si los métodos de la clase base que han sido sobrescritos son declarados como virtuales.
- 4.- Si los métodos virtuales son ejecutados a través de un puntero a la clase base.

Entonces estaremos ante lo que en Programación orientada a objetos se denomina:

POLIMORFISMO



Pasemos al programa

CLASE ABSTRACTA

Definición:

Muchas veces nos encontramos en la situación en la que tenemos varias clases que no está relacionadas pero que tienen métodos con encabezados iguales.

Se quiere trabajar esos métodos empleando polimorfismo, pero no existe una jerarquía de clases.

Para solucionar este problema se crea una clase que declare estos métodos de manera virtual y se hace que las demás clases hereden de ésta.

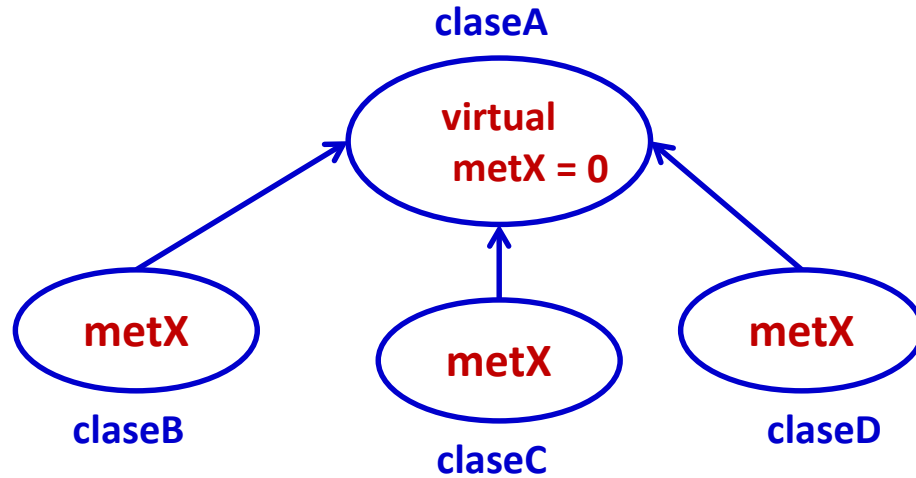
Método virtual puro:

Un método virtual puro es aquel que no se requiere que tenga implementación.

Para evitar que el compilador detecte un error de enlace, se le implementa de la manera siguiente:

```
virtual T metodoA (...) = 0;
```

La igualdad con cero hará que el compilador no busque la implementación.



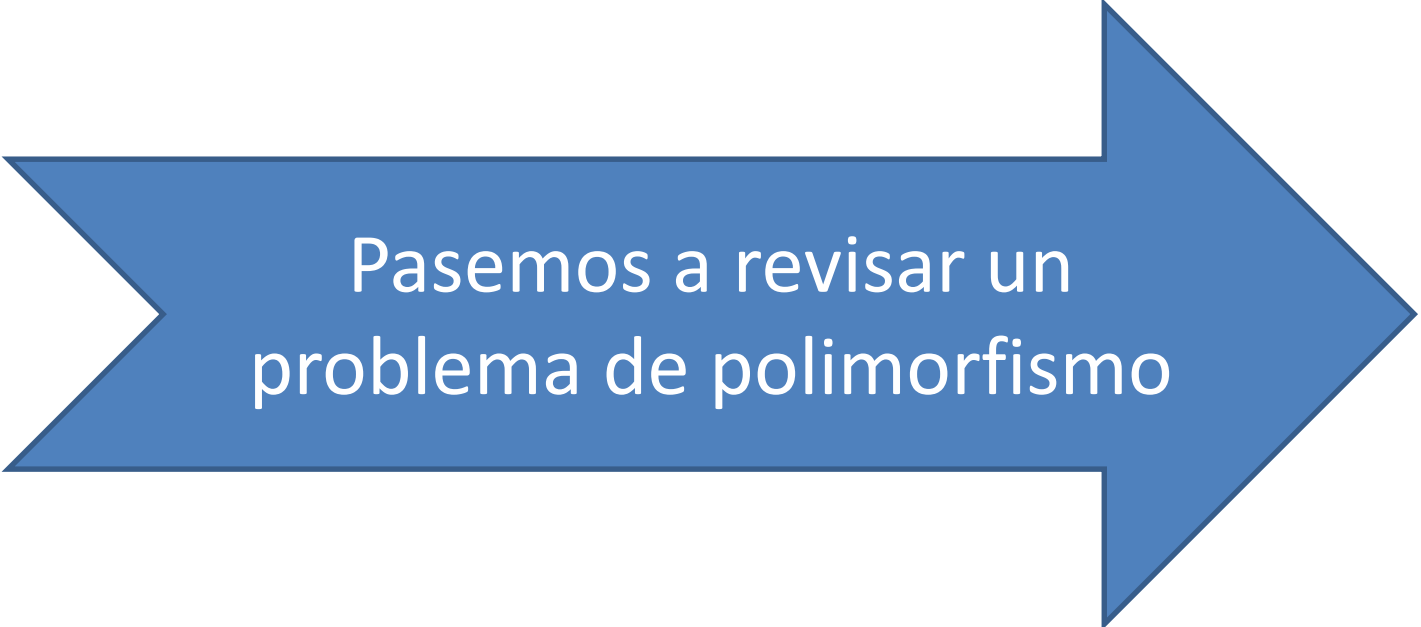
Como la nueva clase solo se ha creado para poder realizar polimorfismo, se declaran los métodos **puros**.

Definición:

Cuando una clase define por lo menos un método virtual puro, la clase recibe el nombre de **“CLASE ABSTRACTA”**.

Restricción:

Una clase abstracta no puede ser instanciada. Esto es, no se pueden declarar objetos de esa clase.



Pasemos a revisar un
problema de polimorfismo

Planteamiento del problema de las fichas:

- Hay que hacerlo con **polimorfismo**, entonces habra que definir:
 - Herencia
 - Métodos virtuales
 - Métodos sobrescritos
 - Punteros a clase base

- Tenemos tres tipos de fichas: A , B y C.
- Todas tienen los mismos atributos: identificación, fila y columna
- Como no vamos a repetir los atributos en cada ficha, se definirá una clase base que defina y maneje los atributos.

Clase Base: "Ficha"

Clase
Abstracta

Atributos:

"Identificación"

"Fila"

"Columna"

Métodos:

¿Virtual? → "Métodos selectores" ⊗

¿Virtual? → ✗ "Mover ficha" = 0 ✓

¿Virtual? → ✗ "Imprimir ficha" = 0 ✓

Clases Derivadas: “A”, “B” y “C”

Atributos:

Métodos:

“Mover ficha”

“Imprimir ficha”



Pasemos al programa

CLASES AUTOREFERENCIADAS

Una clase autoreferenciada, al igual que las estructuras autoreferenciadas son clases que pueden enlazarse a otros elementos similares de modo que podamos formar con ellas: “listas ligadas”, “arboles”, “pilas”, “colas”, etc.

El problema con las clases es que debemos salvaguardar siempre el encapsulamiento de las clases involucradas.

Esto quiere decir que no podemos proporcionar al usuario el control de la lista, devolviendo punteros a los nodos o a los datos.

Entonces si o tenemos la posibilidad de acceder directamente a los punteros ni a los datos.

¿Cómo implementamos una lista ligada o un árbol, si el acceso a estos elementos y su enlace se realiza por medio de punteros?

El diseño que he elaborado permitirá, de una manera muy sencilla, crear estas estructuras.

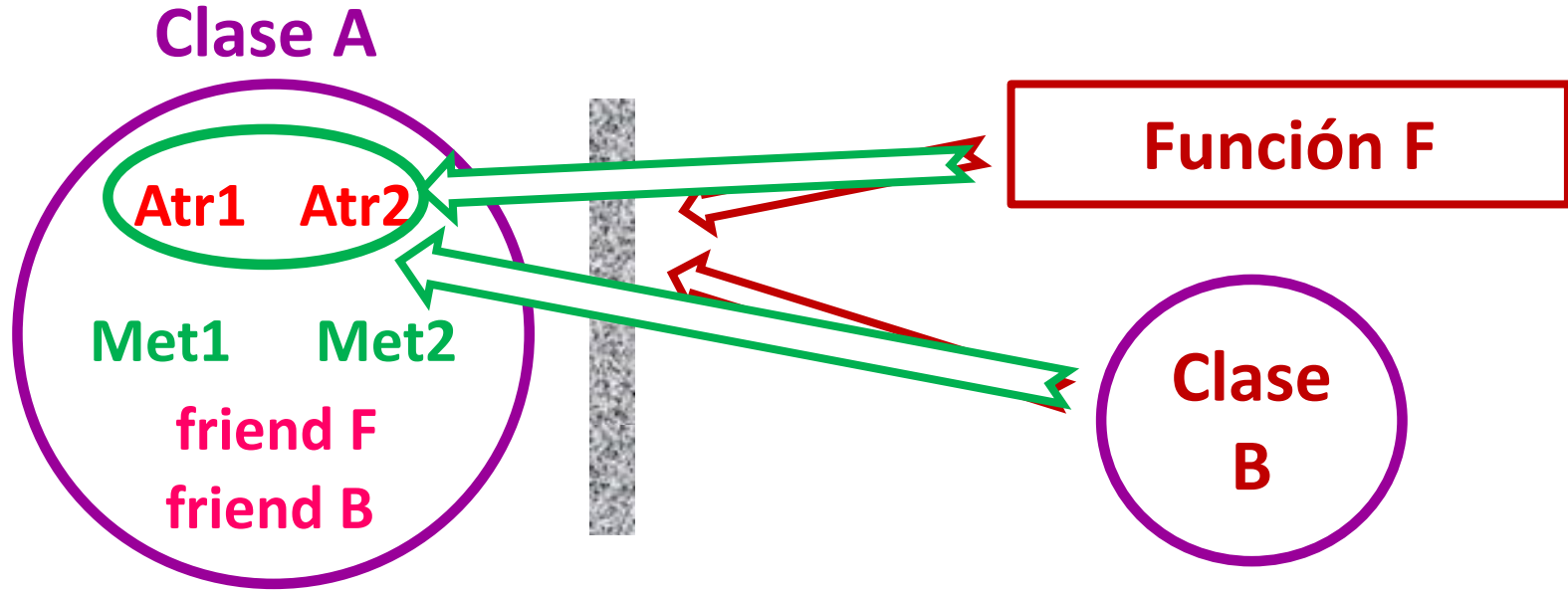
Nos serviremos para este fin de una herramienta propia de C++ denominada “**clausula friend**”.

Definición: Clausula “friend”

La clausula “friend” es una herramienta propia del lenguaje C++ que se incorpora a una clase y que permite dar a otras clases o funciones independientes acceso total a sus elementos, incluso a aquellos que se encuentran en la zona privada.

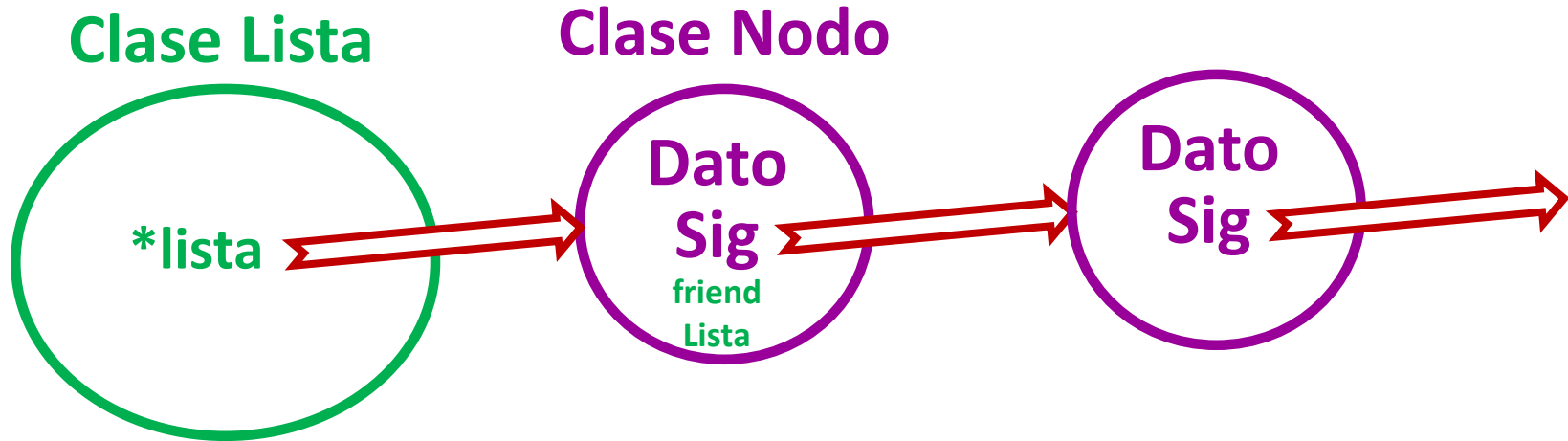
La clausula friend, así como la zona protected, puede muy perjudicial para una aplicación orientada a objetos, sin embargo, empleada de la forma como he diseñado estas estructuras, se logra implementar en forma sencilla estas estructuras.

Implementación: Clausula “friend”



La clase A declara amiga a la función F. La función F NO puede declararse amiga de la clase.

Implementación : Listas enlazadas



Lo que conseguimos con esto es simular que los elementos de `Nodo` están definidos en `Lista`.



Pasemos al programa

Implementación : Listas enlazadas

