I102 - Paradigmas de Programación

OOP - POLIMORFISMO

Prof. Mariano Scaramal

Ingeniería en Inteligencia Artificial

Que conceptos de C++?

- Conceptos
  - Concepto de Polimorfismo
  - Static y Dynamic Binding
  - Polimorfismo y Ejemplo
  - El Destructor Virtual
  - Interfaces y Clases Abstractas
  - Ejemplo de D&D
  - Operadores *static_cast y dynamic_cast,* el Calificador *final.*

Que es Polimorfismo?

La palabra *polimorfismo* deriva del griego, donde significa "*muchas formas*".

En OOP, *polimorfismo* significa que el objeto puede tomar distintas formas, en particular, permite que objetos de diferentes clases sean tratados como objetos de la clase base que tienen en común.

Esto permite tener ventajas como:

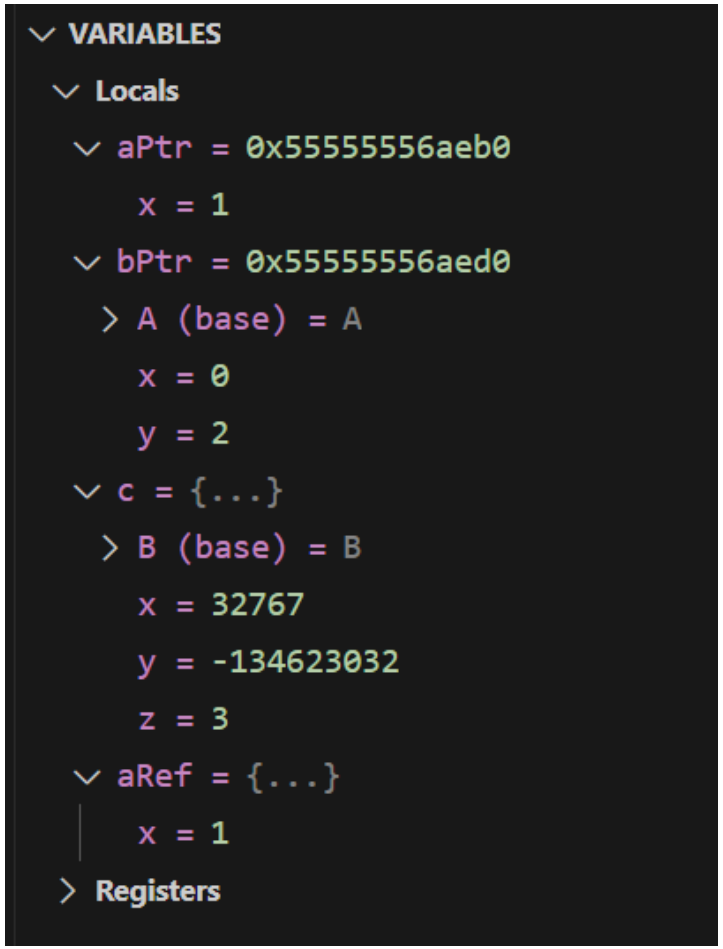
- Generalización de comportamientos: la superclase, con un comportamiento genérico, deberá ser reemplazado por el comportamiento de cada subclase, asegurándose la implementación de este comportamiento.
- Reducir dependencias entre componentes: permite tener código que sea independiente de implementaciones específicas.
- Flexibilidad en tiempo de ejecución: como todas las subclases pueden ser tratados como una clase base, un contenedor tipado como la clase base podrá contener las distintas subclases en él.

Casting un Objeto a su Clase Base

Un objeto puede ser casteado a su clase base (downcasting) mediante el uso de un puntero o una referencia (lo más usual es usar punteros).

Ejemplo:

```
A* aPtr = new C(1, 2, 4);
B* bPtr = new C(7, 2, 1);
C c(1,2,3);
A & aRef = c;
```



A la derecha se pueden ver las variables obtenidas.

En el código de la página siguiente, la clase A tiene solamente la variable *x,* la clase B las *x* e *y,* mientras que la clase C tiene *x*, *y*, y *z.*

Con esta información, se ve que los objetos creados fueron casteados a la clase base.

Casting un Objeto a su Clase Base (cont'd)

```
#include <iostream>

class A{
public:
    int x;
    A(int x) : x(x){}
};

class B : public A{
public:
    int x, y;
    B(int x, int y) : A(x), y(y){}
};
```

```
class C : public B{
public:
    int x, y, z;
    C(int x, int y, int z) : B(x,y), z(z){}
};

int main(){
    A* aPtr = new C(1, 2, 4);
    B* bPtr = new C(7, 2, 1);
    C c(1,2,3);
    A & aRef = c;

    return 0;
}
```

Static Binding

Static binding es la forma default por el cual un método/función es invocado. Se usa cuando no se necesita polimorfismo.

También se lo conoce como *Método No Polimórfico*, debido a que el método/función a ejecutar es determinado en tiempo de compilación mediante el tipo con el que es declarado el objeto/función.

Aquí, *animalPtr* es declarado como puntero a *Animal.*

Al ejecutar el método *speak()*, se ejecutará el método de la clase *Animal, d*ando:

Soy Sir Animal

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

class Animal {
public:   // Static binding
   void speak() { cout << “Soy Sir Animal!" << endl;}
};

class Dog : public Animal {
public:
   void speak() { cout << "Guau, guau!" << endl; }
};

int main() {
   Animal* animalPtr;
   Dog dog;
   animalPtr = &dog;
   // El animal sigue hablando como animal
   animalPtr->speak();
   return 0;
}
```

Dynamic Binding

Dynamic binding permite resolver que método ejecutar en tiempo de ejecución utilizando el tipo del objeto y no el de la declaración.

En el ejemplo de la página siguiente, la declaración de *animalPtr* se realiza con la clase *Animal*, pero los métodos que se ejecutan son los de los objetos que se pasan: *dog* y *cat.*

Los métodos de la clase base deben usar la palabra reservada ***virtual*** para usar *dynamic binding* y para que los métodos puedan ser sobreescritos (overriden) por los de la clase derivada.

El uso de herencia y de la palabra reservada ***virtual*** (que activa el dynamic dispatch) es lo que permite la implementación de polimorfismo.

Dynamic Binding (cont'd)

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

class Animal {
public: // Metodo virtual
   virtual void speak() { cout << "Soy Sir Animal" << endl; }
};

class Dog : public Animal {
public: // Sobreescribe speak de animal con el de perro.
   void speak() override { cout << "Guau y Bau" << endl; }
};

class Cat : public Animal {
public: // Sobreescribe speak de animal con el de gato.
   void speak() override { cout << "Miau y Mow!" << endl; }
};
```

```
int main() {
   Animal* animalPtr;
   Dog dog;
   Cat cat;

   animalPtr = &dog;
   // Dynamic binding => ladra,
   animalPtr->speak();

   animalPtr = &cat;
   // Dynamic binding => maulla.
   animalPtr->speak();

   return 0;
}
```

Polimorfismo

Mediante herencia y dynamic binding se obtiene la implementación de polimorfismo.

Una vez un método es declarado *virtual*, este lo será durante toda la cadena de herencia hasta el final.

Si una clase no sobreescribe un método virtual de su función base, la clase derivada simplemente hereda la implementación de la clase base.

Polimorfismo permite crear un programa que es general y dejar que el dynamic dispatch se encargue de lo específico (que método llamar).

Al mismo tiempo, ayuda a la extensibilidad del software. Esto se logra porque nuevas subclases pueden ser creadas para utilizar el mismo software que usan otras subclases.

Polimorfismo - Ejemplo

```
#include <iostream>
using namespace std;
class Enemy {
public:
   virtual void attack(){}
};

class Goblin : public Enemy {
public:
   void attack() override {
      cout << "Goblin atacan con mazo! -5 HP" << endl;
   }
};

class Skeleton : public Enemy {
public:
   void attack() override {
       cout << “Esqueleto ataca con espada! -10 HP" << endl;
   }
};
```

```
class Dragon : public Enemy {
public:
   void attack() override {
      cout << "Dragon ataca con fuego! -50 HP" << endl;
   }
};

int main() {
   Enemy* enemies[] = { new Goblin(),
                        new Skeleton(), new Dragon() };

   for (int i = 0; i < 3; i++) // Comienza ataque enemigo
      enemies[i]->attack();

   for (int i = 0; i < 3; i++)  // Se eliminan los punteros
      delete enemies[i];
   return 0;
}
```

Destructor Virtual

Que ocurriría con el destructor si no se utiliza dynamic binding?

Como se está borrando *base*, que es un objeto del tipo puntero a *Base*, se verá la leyenda de que se ha ejecutado el destructor de *Base*, pero no el de *Derived*.

Si se borraría *der*, se haría en forma correcta

Imprime:

Constructing base
Constructing derived
Destructing base

```
#include <iostream>
class Base {
public:
   Base() { std::cout << "Constructing base\n"; }
   ~Base() {std:: cout<< "Destructing base\n"; }
};

class Derived: public Base {
public:
   Derived() {std:: cout << "Constructing derived\n"; }
   ~Derived() {std:: cout << "Destructing derived\n"; }
};

int main() {
   Derived *der = new Derived();
   Base *base = der;
   delete base;

   return 0;
}
```

Destructor Virtual (cont'd)

Aquí, simplemente se agregó la palabra ***virtual*** al destructor.

En este caso, al borrar el objeto *base*, se llamará primero al destructor de la clase *Derivada* y luego al de la clase *Base*. Esto ocurre si se borra *der* o *base*.

Imprime:

Constructing base
Constructing derived
Destructing derived
Destructing base

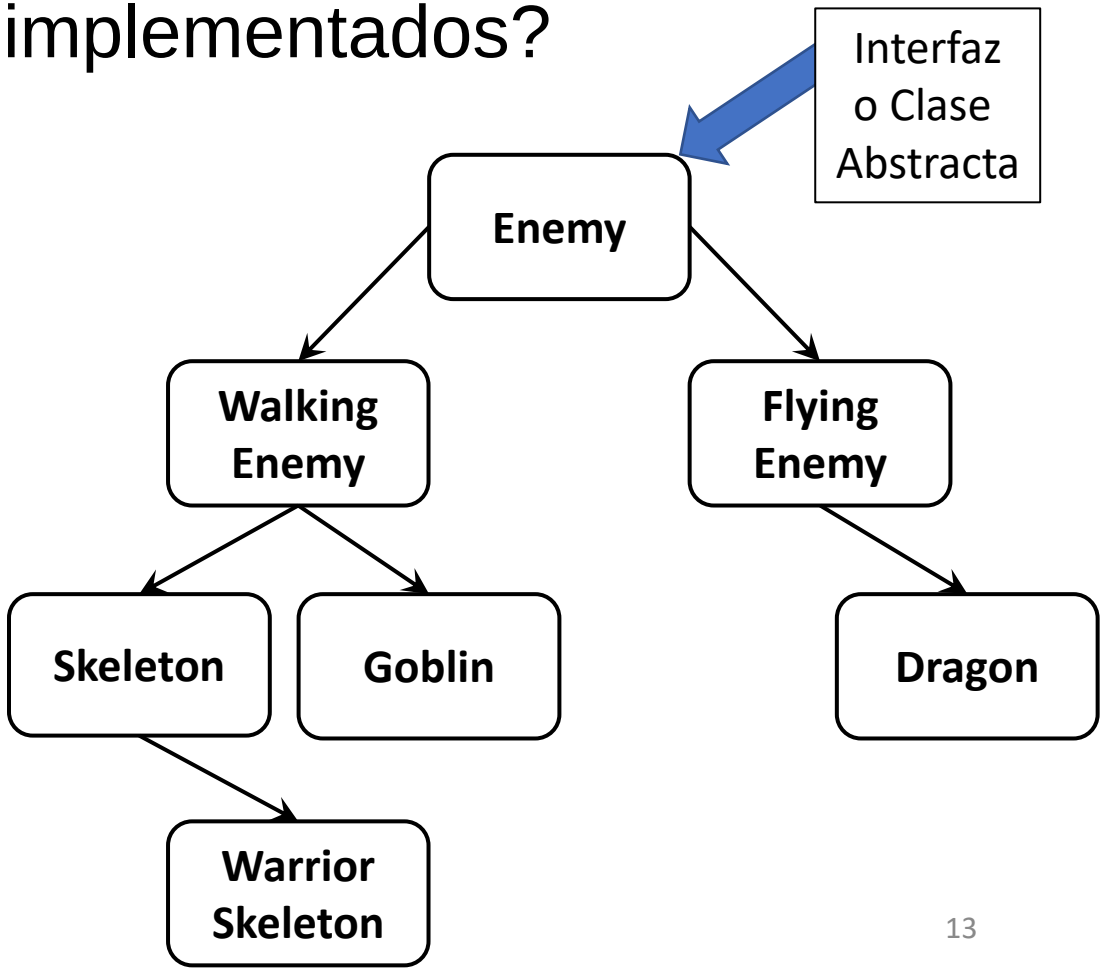
```
#include <iostream>
class Base {
public:
   Base() { std::cout << "Constructing base\n"; }
   virtual ~Base() {std:: cout<< "Destructing base\n"; }
};

class Derived: public Base {
public:
   Derived() {std:: cout << "Constructing derived\n"; }
   ~Derived() {std:: cout << "Destructing derived\n"; }
};

int main() {
   Derived *der = new Derived();
   Base *base = der;
   delete base;

   return 0;
}
```

Interfaces y Clases Abstractas

El concepto de generalizar clases es permitido por la herencia y el polimorfismo. Pero la pregunta que surge es: como hacer para asegurarse que ciertos métodos sean implementados?

Volviendo al ejemplo de los enemigos atacando, como asegurarse de que el programador implemente para cada nuevo enemigo su modo particular de ataque?

La forma de hacer esto es mediante interfaces y clases abstractas.



Interfaces

Una interfaz o clase abstracta pura es escrita como una clase (pero no lo es) donde todos sus métodos son funciones virtuales puras (deberán ser implementadas en clases derivadas).

Se las puede ver como un contrato donde se establecen las normas de implementación para las clases que derivan de esta.

Las interfaces no pueden instanciarse, pero al ser heredadas, sus métodos virtuales **deben** (por ser virtuales puros) ser implementados.

Dado que las interfaces deben ser usadas como superclases, se suelen utilizar para herencia múltiple. Así, una clase deberá implementar los métodos de todas las interfaces de las cuales es derivada.

Clases Abstractas

Es una clases donde algunos de sus métodos son funciones virtuales puras. Estas clases no pueden instanciarse, pero al ser heredadas, sus métodos virtuales deben ser implementados.

Se la utiliza cuando hay datos o comportamientos comunes entre todas las clases derivadas de esta clase.

Una clase abstracta puede ser una clase derivada de una interfaz en donde no es necesario implementar los métodos de la interfaz (incluso se pueden agregar más funciones virtuales puras).

Si una clase abstracta es derivada de una interfaz y luego esta se convierte en una clase base de una clase, esta clase deberá implementar todos los métodos virtuales puros de la interfaz y de la clase abstracta.

Métodos Virtuales Puros

Un método virtual puro es definido de la siguiente manera:

virtual void doSomething() = 0;

Cuando se define un método de esta manera, la clase se convierte en una clase abstracta o una interfaz, por lo que no podrá ser instanciada.

Los métodos virtuales sólo podrán ser definidos en las clases derivadas y no en las clases base.

La diferencia entre los métodos virtuales y los virtuales puros son:

Un método virtual tiene una implementación en la clase que lo define y da a las clases derivadas la opción de sobreescribirlo (override) con su propia versión,

Un método virtual puro no tiene una implementación en la clase que lo define y debe ser sobreescrito (override) en la clase derivada.

Ejemplo – Juego Tipo D&D - Header

```
#pragma once

class Enemy {
public:
    Enemy(int hp) : hp(hp){};
    virtual int getHP() = 0;      // Todos tienen HP,
    virtual bool isDead() = 0; // pueden morir
    virtual int attack() = 0;     // y atacar
    virtual ~Enemy();
protected:
    int hp;                  // Puntos de vida
    bool dead = false;   // Por default esta vivo
};
```

Enemy es una clase abstracta. Las interfaces no deberían tener ni atributos ni constructor ni destructor.

```
// "Override" se usa al reemplazar un método
//virtual de la clase base.
class WalkingEnemy : public Enemy {
public:
    WalkingEnemy(int hp) : Enemy(hp){};
    int getHP() override;       // Devuelve los puntos de vida
    bool isDead() override;   // Esta muerto el enemigo?
    int attack() override;       // Ataca restando hp
    virtual int boneAttack() = 0; // Ataque especial
    void walk();                        // Se desplaza caminando
    virtual ~WalkingEnemy();
};

class Skeleton : public WalkingEnemy {
public:
    Skeleton(int hp) : WalkingEnemy(hp){};
    int attack() override;     // Hace el ataque del esqueleto
    int boneAttack() override; // Hace el ataque especial
    ~Skeleton();
};
```

Ejemplo – Juego Tipo D&D - Source

```
#include <iostream>
#include "enemies.h"
// No se implementa nada de la clase base Enemy
int WalkingEnemy::getHP() {
    return hp;
} // Getter de HP

bool WalkingEnemy::isDead() {
    return (hp <= 0 ? true : false);
} // Si no tiene mas HP esta muerto

int WalkingEnemy::attack() {
    return 0;
} // Por default no remueve HPs

void WalkingEnemy::walk() {
    std::cout << "Is walking!!" << std::endl;
} // El enemigo camina, no vuela
```

```
int Skeleton::attack() {
    return -5;
} //El esqueleto ataca con -5 por default

int Skeleton::boneAttack() {
    return -10; // Bone attack hace -10
}

Skeleton::~Skeleton(){
    std:: cout << "Destructing Skeleeton\n";
}

WalkingEnemy::~WalkingEnemy() {
    std::cout << "Destructing WalkingEnemy\n";
}

Enemy::~Enemy() {
    std::cout << "Destructing Enemy\n";
}
```

Ejemplo – Juego Tipo D&D - Main

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

int main(){
   Skeleton sk1 = Skeleton(20); // El esqueleto tiene 20 HP
   // Verificando puntos de vida
   std::cout << "El esqueleto tiene " << sk1.getHP()
   << " puntos de vida." << std::endl;

   //Verificando si esta vivo
   if (sk1.isDead())
      std::cout << "El esqueleto esta muerto" << std::endl;
   else // En realidad siempre esta muerto, es un esqueleto!
      std::cout << "El esqueleto esta vivo" << std::endl;

   std::cout << "Prueba terminada" << std::endl << std::endl;
   return 0;
}
```

La salida del programa será:

```
El esqueleto tiene 20 puntos de vida.
El esqueleto esta vivo
Prueba terminada

Destructing Skeleeton
Destructing WalkingEnemy
Destructing Enemy
```

Siempre conviene verificar en una prueba que se estén destruyendo correctamente las clases.

Los Calificadores de Métodos *default* y *delete*

El calificador *default* se utiliza para explícitamente definir el constructor y/o destructor default. No hace más que pedir que se cree el des/cons-tructor vacío.

El calificador *delete*, deshabilita los métodos. Usualmente esto se hace para evitar copias o asignaciones.

```
class Drone {
public:
   // Constructor default generado explícitamente por el
   //compilador
   Drone() = default;
   // Deshabilita el copy constructor evitando copias
   Drone(const Drone&) = delete;
   // Deshabilita el operador de asignación de copia para
   //prevenir asignaciones
   Drone& operator=(const Drone&) = delete;
   // Indica explicitamente al compilador que cree el
   //destructor
   ~Drone() =  default;
   void show() {
      std::cout << "Objeto Drone creado!" << std::endl;
   }
};

int main() {
   Drone d1;

   // Drone d2 = d1;  // ERROR: No hay un Copy constructor
   // Drone d2(d1);    // ERROR: No hay un Copy constructor
   // d1 = d2;   // ERROR: No hay un operador de asignación
   d1.show();
}
```

Ejemplo de Interfaz

Una interfaz no define un constructor por defecto. Ni siquiera el constructor.

Tampoco es necesario definir un destructor en la interfaz, se puede hacer en la clase.

No obstante, es posible hacerlo en la interfaz como un método virtual puro y luego implementarlo para la interfaz.

```
class IEnemy {
public:
   virtual int getHP() = 0;
   virtual bool isDead() = 0;
   virtual int attack() = 0;
   virtual ~IEnemy() {} // Destructor virtual
};
```

```
class IEnemy {
public:
   virtual int attack() = 0;
   virtual ~IEnemy() = 0;  // Destructor virtual puro
};

IEnemy::~IEnemy(){ // El destructor debe ser implementado
      std::cout << "El destructor de IEnemy" << std::endl;};

class A : public IEnemy{
public:
   A(){std::cout << "El constructor de A" << std::endl;}
   int attack() {return 1;}
   ~A() {std::cout << "El destructor de A" << std::endl;}
};

int main(){
   A a; // Imprime: “El constructor de A”
   return 0;
}// Imprime: “El destructor de A”, “El destructor de Ienemy”
```

El Operador *static_cast*

Este operador es utilizado para castear en forma explícita entre tipos en tiempo de compilación. Realiza el *cast* en forma segura, aunque no es seguro entre punteros a clases bases y punteros a clases derivada (no es *type safe downcasting* con polimorfismo).

La forma en que se usa es:

static_cast<Tipo>(expresión)

Donde:

- <Tipo> es el nuevo tipo al que se quiere caster.
- (expresión) es la variable u objeto a convertir.

Si no hay polimorfismo se puede usar el static_cast para downcasting, es más rápido que *dynamic_cast*.

El Operador *static_cast (cont'd)*

Este operador se puede utilizar de distintas formas:

- Entre variables numéricas:

```
int i = 1;
double d = static_cast<double>(i);
```

- Entre objetos base y derivado:

```
Base* base = new Derived();
Derived* derived = new Derived();
Derived* der = static_cast<Derived*>(base); // Convierte de Base* a Derived*
Base* bs = static_cast<Base*>(derived);      // Convierte de Derived* a Base*
```

- Entre **void* y un puntero específico (ya visto en clases pasadas):

```
void* ptr = malloc(sizeof(int));
int* intPtr = static_cast<int*>(ptr);
```

El Operador *dynamic_cast*

Este operador es utilizado para castear en forma segura objetos polimórficos, es decir, objetos con al menos un método virtual. Principalmente es utilizado para hacer downcasting (objeto base a objeto derivado).

La forma en que se usa es:

dynamic_cast<Tipo>(expresión)

Donde:

- <Tipo> es el nuevo tipo al que se quiere caster.
- (expresión) es la variable u objeto a convertir.

Es más lento que static_cast porque utiliza RTTI (Run-Time Type Information) para conocer los tipos en tiempo de ejecución.

El Operador *dynamic_cast* - Downcasting

En este caso, el código muestra el uso de *dynamic_cast* para un realizar un downcast en la función *main*.

Esta conversión de realiza en tiempo de ejecución por lo que si no es posible realizar el cast, se devolverá un *nullptr*, sino se devolverá un puntero válido.

```
#include <iostream>
class Base {
public:  // Método virtual
   virtual void show() { std::cout << "Clase Base\n"; }
};
class Derivada : public Base {
public:
   void show() override { std::cout << "Clase Derivada\n"; }
};

int main() {  // Puntero a clase Base a objeto Derivada
   Base* b = new Derivada();
   // Downcast seguro
   Derivada* d = dynamic_cast<Derivada*>(b);
   if (d)
      d->show();  // Imprime: Clase Derivada
   else
      std::cout << "Cast failed!\n";
   delete b;
}
```

El Operador *dynamic_cast* - Referencias

En el caso en que el *cast* se haga con una referencia, se devuelve un error, no nullptr, pero se podrá capturar.

El error que se devuelve es del tipo *std::bad_cast*, que puede ser capturado con un *try/catch*.

Este es un mecanismo muy poderoso para manejar un safe downcasting en tiempo de ejecución.

```
#include <iostream>
#include <typeinfo>
class Base {
public: // Como método virtual requerido se usa el Destructor
   virtual ~Base() {}
};
class Derived : public Base {};

int main() {
   Derived d;
   Base& b = d;  // Referencia a de clase Base a objeto Derivada

   try { // Downcast seguro
      Derived& d_ref = dynamic_cast<Derived&>(b);
      std::cout << "Se logro hacer el Cast!\n";
   } catch (const std::bad_cast& e) {
      std::cout << "El cast fallo: " << e.what() << "\n";
   }
   return 0;
}
```

El Calificador *final* Aplicado a Clases

Cuando el calificador *final* es utilizado en una clase, dicha clase no podrá ser heredada por otra clase.

Suele utilizarse para evitar modificaciones en el árbol de herencia de clases, forzar un diseño determinado o por seguridad.

Debido a que la clase *Derivada* esta como *final,* no se podrá usar como una clase base. De hacerlo, se tendrá un error de compilación.

```
#include <iostream>
class Base {
public:
   virtual void show() { std::cout << "Clase Base\n"; }
};

class Derivada final : public Base {
public:  // 'final' no permite producir más clases por herencia
   void show() override { std::cout << "Clase Derivada\n"; }
};

class ProxClase : public Derivada {
public: //Error: a final class type cannot be used as a base class.
   void show() override { std::cout << "Proxima clase\n"; }
};

int main() {
   Derivada d;
   d.show();
}
```

El Calificador *final* Aplicado a Métodos

Cuando el calificador *final* es utilizado en un *método virtual* (sólo tiene sentido en estos métodos) este evita que el método sea sobreescrito.

En el ejemplo, el error de compilación se obtendrá debido a que se utilizó el calificador *final* en el método virtual show().

En este caso, *final* evita que se cambie la funcionalidad provista en *show* y evitando también el generar nuevo código.

```
#include <iostream>

class Base {
public:  // 'final' previene que la clase sea sobreescrita
    virtual void show() final { std::cout << "Clase Base\n"; }
};

class Derivada : public Base {
public:
    // Error: Cannot override 'show' because it's final in Base
    void show() override { std::cout << "Clase Derivada\n"; }
};

int main() {
    Base b;
    b.show();
}
```

Universidad de SanAndrés

Preguntas?