POLIMORFISMO

Por: Yolanda Martínez Treviño
Ma. Guadalupe Roque Díaz de León

Polimorfismo

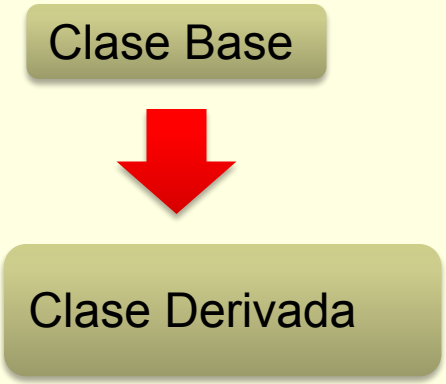
- Llegamos ahora a los conceptos más sutiles de la programación orientada a objetos.
- La virtualización de funciones permite implementar una de las propiedades más potentes de POO: el polimorfismo. Pero vayamos con calma...
- En una clase derivada se puede definir una función que ya existía en la clase base, esto se conoce como "**overriding**".
- La definición de la función en la clase derivada oculta la definición previa en la clase base.
- En caso necesario, es posible acceder a la función oculta de la clase base mediante su nombre completo:
  - <objeto>.<clase_base>::<método>;

Poliformismo

- Por fin vamos a introducir un concepto muy importante de la programación orientada a objetos: el polimorfismo.
- En lo que concierne a clases, el polimorfismo en C++, llega a su máxima expresión cuando se usan junto con apuntadores.
- C++ permite acceder a objetos de una clase derivada usando un puntero a la clase base. En esa capacidad es posible el polimorfismo.
- Pero🥺, sólo podremos acceder a datos y funciones que existan en la clase base, los datos y funciones propias de los objetos de clases derivadas serán inaccesibles.🔐

Apuntador de la clase base apuntando a objeto de la clase derivada.

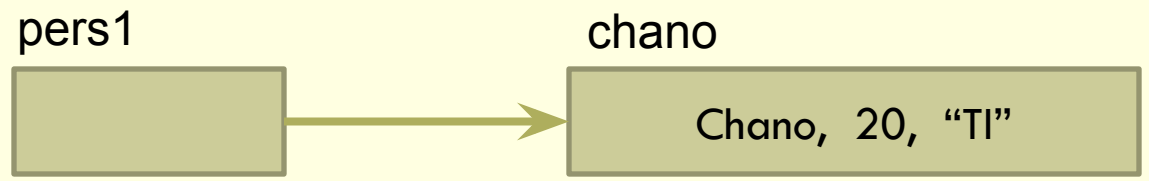
- Es posible tener variables de tipo apuntador para almacenar objetos.
  - Para declarar un apuntador se utiliza el siguiente formato:
    tipo *nombreVariable;
  - Una variable que puede guardar una dirección de memoria se llama apuntador.
- Un objeto de la clase derivada es un objeto de la clase base (pero con más atributos y métodos)
- Una variable apuntador de la clase base puede apuntar a un objeto de la clase derivada.



Apuntador de la clase base apuntando a objeto de la clase derivada.

Ej:

```
Persona   *pers1;
Estudiante  chano("Chano", 20, "TI");
pers1 = &chano;
```



Ejemplo: Clase Estudiante

```
#include <iostream>
#include <string>
using namespace std;
#include "Persona.h"
class Estudiante : public Persona
{
public:
      Estudiante();
      Estudiante(std::string, int,
      std::string);
      std::string getCarrera();
      void setCarrera(std::string);
      std::string str();
  private:
      std::string carrera;
};
```

```
Estudiante::Estudiante(string nom, int ed, string
      ca) : Persona(nom, ed){
  carrera = ca;
}

string Estudiante::getCarrera(){
  return carrera;
}

void Estudiante::setCarrera(string ca){
  carrera = ca;
}

string Estudiante::str(){
 return "\nNombre: " + nombre + "\nEdad: " +
      to_string(edad) + "\nCarrera: " + carrera;

}
```

https://repl.it/@roque_itesm_mx/Poliformismo

Ejemplo

```
#include "Estudiante.h"
#include "Medico.h"
int main() {

  // Una variable que puede guardar una dirección de memoria se
  // llama apuntador. Sintaxis: tipo *ptrNombre;
  Persona *ptrPersona, chanoPersona("Chano", 80);
  Estudiante *ptrEstudiante, chaveloEstudiante("Chavelo", 90, "Dr.");
  // El operador & se utiliza para obtener la dirección de una
  // variable, usa la siguiente sintaxis:
  ptrPersona = &chanoPersona;
  ptrEstudiante = &chaveloEstudiante;
  cout << "\nPtrPersona(chanoPersona)->str():" << ptrPersona->str() << endl;
  cout << "\nPtrEstudiante(chaveloEstudiante)->str():" << ptrEstudiante->str() << endl;
  // Ahora el apuntador persona se le asigna la dirección de un objeto estudiante
  // sin embargo el objeto al almacenarlo en un apuntador de la clase Base - este
  // objeto se comporta como si fuera de la clase Base-
  //La funcionalidad depende del  tipo de apuntador, no del tipo del objeto -
  // Por lo tanto solo se podrán llamar a los métodos de la clase Persona
  ptrPersona = &chaveloEstudiante;
  cout << "\nPtrPersona(chaveloEstudiante):" << ptrPersona << endl;
  cout << "\nPtrPersona(chaveloEstudiante)->str():" << ptrPersona->str() << endl;
```

objeto chaveloEstudiante guardado en el apuntador de tipo Persona ptrPersona ejecutó el método str() de la clase

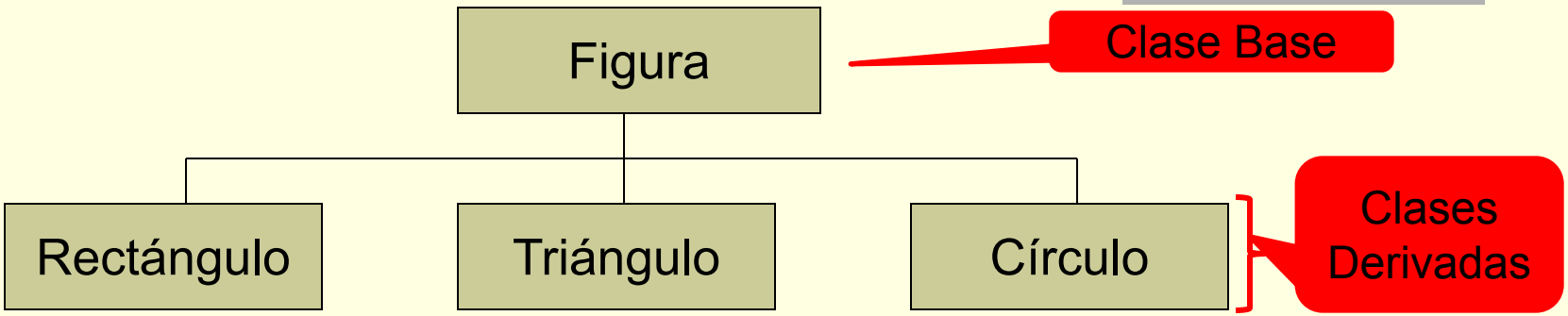
En resumen

- Al almacenar un objeto de la clase derivada en un apuntador a la clase base, el objeto se comporta como si fuera de la clase base, por eso se ejecutó el método str( ) de la clase Persona.

- Es decir, la funcionalidad depende del tipo del apuntador, no del tipo del objeto.

- Entonces como el apuntador es a un objeto de la clase Persona, solo se puede hacer referencia a los datos y llamar a los métodos que se heredaron de la clase Persona.

Funciones Virtuales



- Supón que tenemos una aplicación que permite dibujar figuras en la pantalla, las figuras pueden ser Rectángulos, Círculos, Triángulos, etc.
- En la aplicación queremos un arreglo de figuras, y para dibujar solo tuviera que mandar llamar el método dibuja() de cada uno de los objetos almacenados en el arreglo.
- Sería útil que al llamar al método dibuja( ) de cada elemento del arreglo se ejecutara el dibuja( ) del objeto correspondiente, no el dibuja( ) de la clase **Figura**.

Funciones Virtuales

- Pero de acuerdo con lo que vimos en el ejemplo de la clase Persona y Estudiante: si hacemos un arreglo de objetos de tipo Figura, solo se podrá tratar a dichos objetos como Figura, no como Rectángulo, Círculo, etc; por lo que no sería posible que cada objeto del arreglo llamara a su método dibuja( ) que le corresponde, sino llamaría al método dibuja( ) de la clase Figura.

¿Cómo resolver esto? 😳

Para eso existen las funciones virtuales 😊

Funciones Virtuales

- Al usar una función virtual, el tipo del objeto apuntado,, es el que determina cual versión de la función utilizar – (la de la clase base o la de la clase derivada).

- El formato para indicar que una función es virtual es colocar la palabra "virtual" antes del encabezado de la función dentro de la declaración de la clase base.

  Por ejemplo:

  virtual void muestraDatos();

- Revisemos de nuevo el ejemplo de la clase Persona y Estudiante, pero ahora definiendo el método muestraDatos de la clase Persona como virtual.

Ejemplo Funciones Virtuales

Se tomó el ejemplo de la clase Persona y Estudiante y se cambió solamente la declaración del método str( ) de la clase Persona de esta forma:

```
virtual string str();
```

https://repl.it/@roque_itesm_mx/FuncionesVirtuales

Otro ejemplo: Funciones virtuales

```
int main() {
   Persona *ptrPersona, chanoPersona("Chano", 80), chonita;
   Estudiante *ptrEstudiante, chaveloEstudiante("Chavelo", 90, "Dr.");
   Medico gattel("Hugo Lopez",81,"Infectologo",0), manuel;
   Persona *arrPersonas[]={&chanoPersona, &chaveloEstudiante, &chonita, &gattel,
      &manuel};
   // El operador & se utiliza para obtener la dirección de una variable
   ptrPersona = &chanoPersona;
   ptrEstudiante = &chaveloEstudiante;
   cout << "\nPtrPersona(chanoPersona)->str():" << ptrPersona->str() << endl;
   cout << "\nPtrEstudiante(chaveloEstudiante)->str():" << ptrEstudiante->str() << endl;
   ptrPersona = &chaveloEstudiante;
   cout << "\nPtrPersona(chaveloEstudiante)->str():" << ptrPersona->str() << endl << endl;
   for (int iR = 0; iR < 5; iR++)
   {
      cout << arrPersonas[iR]->str() << endl << endl;
   }
      return 0;
}
```

Ahora cada objeto llama al método str() de la clase que corresponde con el tipo del que está definido.

Funciones Virtuales

- Como puedes observar en los ejemplos, al declarar una función virtual, la función a la que se llama depende del tipo del objeto y no del tipo del apuntador.
- Esto puede suceder así gracias al enlace tardío (dinamic binding) que se hace con las funciones virtuales.
- Una nota importante sobre este tema es que si se llama a una función virtual utilizando un objeto que no sea apuntador, se llamará a la función de la clase base.
- Es decir, utilizar una función virtual de manera que cada objeto llame a la función que le corresponda, solamente se puede hacer con variables de tipo apuntador a objetos.

Polimorfismo

El término polimorfismo se refiere al hecho de que se llama el mismo método(nombre) para diferentes objetos y **cada objeto ejecuta el método que le corresponde** (es decir, el método de la clase con la cual fue creado); de la forma en que lo vimos en el ejemplo anterior.

Ejemplo: clase Pers

#ifndef - directiva de compilador que verifica si está definido el nombre y si no está definido lo define usando la directiva de compilador #define.

```
#ifndef ClasePersona
#define ClasePersona
#include <iostream>
#include <string>
using namespace std;
class Persona
{public:
        Persona();
        Persona(string, int);
        string
      getNombre();
        void
      setNombre(string);
        int getEdad();
        void setEdad(int);
        void
      muestraDatos();
protected:
        string nombre;
        int edad;
};

Persona::Persona()
{       nombre = "N/A";
        edad = 0;
}

Persona::Persona(string nom, int ed)
{       nombre = nom;
        edad = ed;
}

string Persona::getNombre()
{       return nombre;
}

void Persona::setNombre(string
      nom)
{       nombre = nom;
}
int Persona::getEdad()
{       return edad;
}
void Persona::setEdad(int ed)
{       edad = ed;
}
void Persona::muestraDatos()
{       cout<<"Nombre:
      "<<nombre<<" edad: "<<edad;
}
#endif
```

Fin del #ifndef