FIEC

FACULTAD DE INGENIERÍA EN ELECTRICIDAD Y COMPUTACIÓN

PROGRAMACIÓN APLICADA A LA AUTOMATIZACIÓN

M.Sc. Alexander Prieto León

Conferencia VIII

Unidad II Programación orientada a objetos y estructuras de datos aplicadas a automatización

2.3 Herencia y polimorfismo (segunda parte)

OBJETIVOS

Objetivos:

- Convertir entre objetos de clases base y derivadas.
- Definir los métodos virtuales.
- Definir las clases base abstractas y los métodos virtuales puros.
- Implementar aplicaciones mediante el uso del polimorfismo.

BIBLIOGRAFÍA

Bibliografía:

- Deitel and Deitel. Como programar en C/C++. Segunda edición o superior.
- García de Jalón, J.; y otros. Aprenda C++ como si estuviera en primero. Universidad de Navarra.

ENCUESTA

- **Realizar la encuesta**: vínculo al comienzo de esta sesión en Teams.

  - PROGRAMACIÓN APLICADA A LA AUTOMATIZACIÓN - PARALELO #2
    https://forms.gle/ohPsHPqtKTv5ChBR8

  - PROGRAMACIÓN APLICADA A LA AUTOMATIZACIÓN - PARALELO #1
    https://forms.gle/KHDg17VD2aNjxE7T6

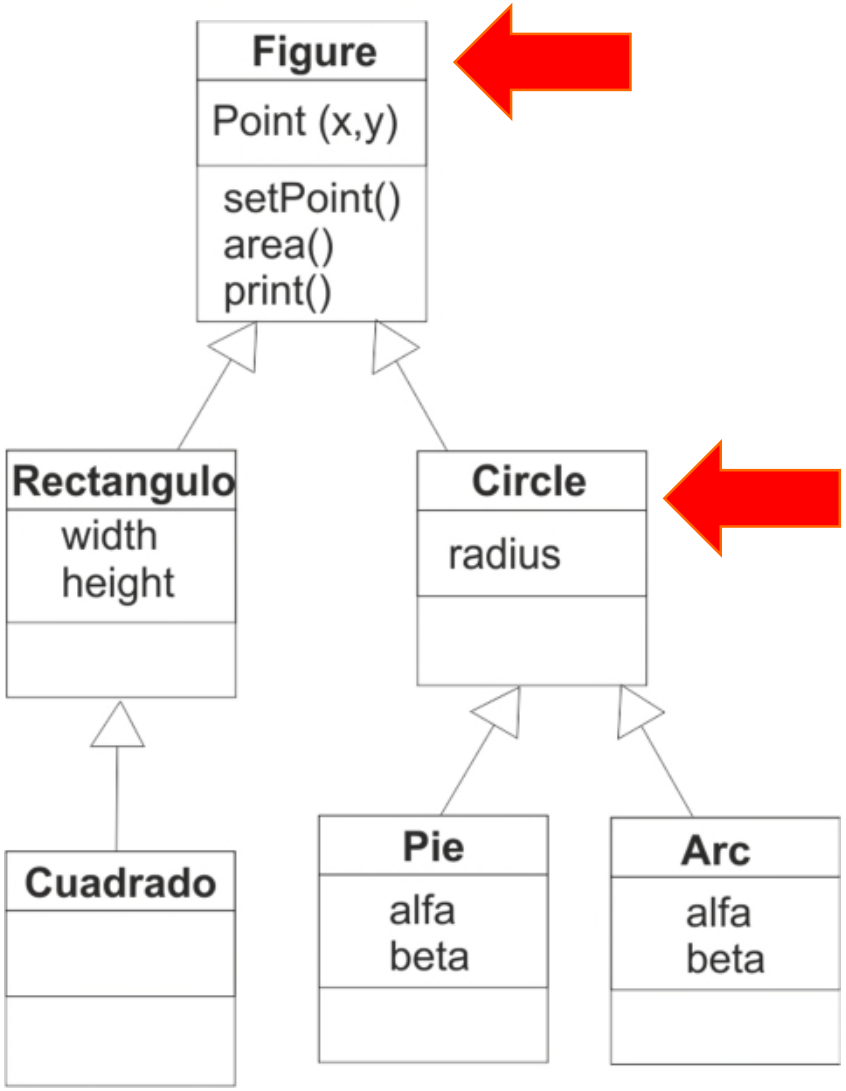
  - Principales comentarios poner en chat de Teams además.

En la clase anterior:

Preguntas iniciales:

- ¿Qué es la herencia?
- ¿Qué implica en control de acceso **protected**?
- ¿Qué elementos no se heredan?
- ¿Cómo se inicializan los elementos de clase base en la clase derivada?
- ¿Cuándo se debe redefinir un método en una clase derivada?
- ¿Cómo se debe llamar a un método de clase base en una clase derivada si:
  - a)dicho método no ha sido redefinido en la clase derivada?
  - b)dicho método ha sido redefinido en la clase derivada?

- **En la clase anterior:**



En la clase anterior:

```
class Figure{
  protected:
            float x,y;
  public:
            Figure(float=0, float=0);
            void setPoint(float, float);
            float getX() const {return x;};
            float getY() const {return y;};
            float area() const;
            void print() const;
 };
```

```
Figure::Figure(float a, float b) {
        x = a;
        y = b;
}
void Figure::setPoint(float a, float b) {
        x = a;
        y = b;
}
float Figure::area() {
        return 0;
}
void Figure::print() {
  cout<<"Point=["<<x<<","<<y<<"]"<<endl;
}
```

En la clase anterior:

```
class Circle : public Figure{
  protected:
     float radius;
  public:
     Circle(float=0, float=0, float=0);
     void setRadius(float r)
     {if (r>=0) radius = r;}
     float getRadius() const { return radius;}
     float area() const;
     float print() const;
 };
```

```
Circle::Circle(float a, float b, float r)
          :Figure(a, b) {
   if (r>=0) radius = r;
}
float Circle::area() const {
       return radius*radius*3.14159;
}
void Circle::print() {
       Figure::print();
       cout<< "Radius = "<< radius << endl;
}
```

En la clase anterior:

```
int main()
{
    Figure *figurePtr, f(3.1, 3.5);
    Circle *circlePtr, c(2.7, 1.2, 8.9);
    figurePtr = &f;
    circlePtr = &c;
    figurePtr ->print();              // imprime sólo el punto
    c.print();                        //imprime punto y radio
    float a= circlePtr->area()        // Área del círculo
    :
    :
}
```

- **Conversión entre clases base y derivada. Explicita e implícita.**
  - Un objeto de la clase derivada pública siempre puede ser tratado como un objeto de la clase base. Lo que no es cierto es lo contrario.
  - Es correcto hacer referencia a un objeto de clase derivada mediante un puntero a clase base. En este caso solo es posible acceder a los miembros de la clase derivada que pertenecen también a la clase base.
  - En el ejemplo anterior sería:

```
figurePtr = &c;
```

Conversión entre clases base y derivada. Explicita e implícita.

- Es incorrecto hacer referencia a un objeto de clase base mediante un puntero a clase derivada. En este caso el apuntador de clase base deberá ser convertido explícitamente a su clase derivada.

```
// circlePtr = &f;   (incorrecto)

// circlePtr = figurePtr; (incorrecto)

circlePtr = (Circlle* )figurePtr; // correcto pero con cuidado. Si realmente
          // figurePtr está apuntando a un objeto Circle no hay problemas.

// float r1=figurePtr->getRadius()  (incorrecto)

float r2=(Circlle* )figurePtr->getRadius() //correcto

float x1= figurePtr->getX()  //correcto

float x2=(Circlle* )figurePtr->getX()  //correcto
```

Conversión entre clases base y derivada. Explicita e implícita.

- Es incorrecto hacer referencia a un objeto de clase base mediante un puntero a clase derivada. En este caso el apuntador de clase base deberá ser convertido explícitamente a su clase derivada.

```
// circlePtr = &f;   (incorrecto)

// circlePtr = figurePtr; (incorrecto)

circlePtr = (Circlle* )figurePtr; // correcto pero con cuidado. Si realmente
          // figurePtr está apuntando a un objeto Circle no hay problemas.

// float r1=figurePtr->getRadius()  (incorrecto)

float r2=(Circlle* )figurePtr->getRadius() //correcto

float x1= figurePtr->getX()  //correcto

float x2=(Circlle* )figurePtr->getX()  //correcto
```

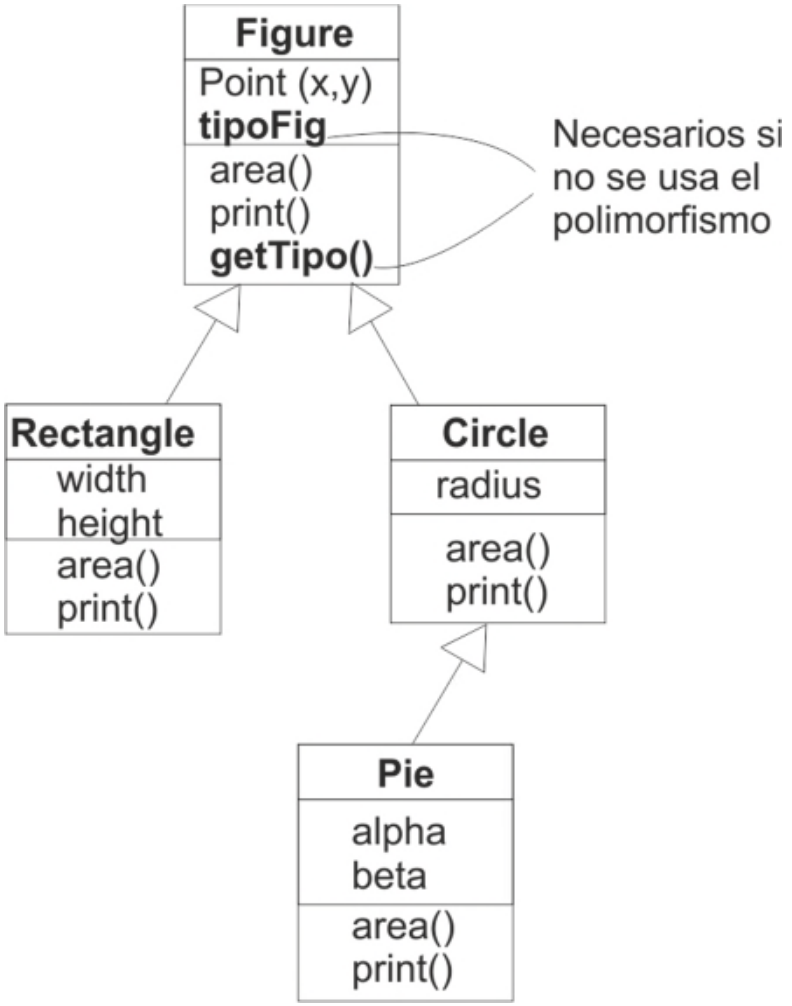
- ***Partiremos de un problema a resolver:***

- Se quiere tener un programa que permita almacenar en un arreglo hasta 20 figuras geométricas que pueden ser Circle, Rectangle o Pie. El usuario deberá entrar las figuras por teclado indicando su tipo y los datos correspondientes. El programa deberá imprimir en pantalla los datos de todas las figuras y sus áreas.

- Nota: considere ya implementadas las clases Rectangle y Pie.

➢ *Problema a resolver:*

```
enum tipo{Figure_,Circle_,Rectangle_,Pie_};


Figure::Figure(float a, float b) {
        x = a;
        y = b;
      tipoFig=Figure_;//Se debe adicionar a cada
                      // constructor asignándole
                     // el tipo de figura
}
```



Problema a resolver:

```
int main()
{
   Figure* Array[20];
   for (int i=0;i<20;i++)
   {
      short tipoF;
      float x,y;
      cout<<"Deme el tipo de figura a introducir"<<endl
      <<"1:Círculo, 2:Rectángulo y 3:Sector  circular"
                                                  <<endl;
      cin>>tipoF;
      cout<<"Deme X y Y"<<endl;
      cin>>x>>y;
      switch(tipoF)
      {
         case 1: cout<<"Deme radio: "<<endl;
                 float r;
                 cin>>r;
                 Array[i]=new Circle(x,y,r);
                 break;
         case 2: cout<<"Deme ancho y alto: "<<endl;
                 float w,h;
                 cin>>w>>h;
                 Array[i]=new Rectangle(x,y,w,h);
                 break;
         case 3: cout<<"Deme radio, alfa y beta: "<<endl;
                 float ra,alp,be;
                 cin>>ra>>alp>>be;
                 Array[i]=new Pie(x,y,ra,alp,be);
                 break;
         default: cout<<"Tipo no reconocido"<<endl;
                  i--;
      }
}
```

- ***Problema a resolver:***

- Observaciones:
  - Para poder introducir los datos siempre se tiene que establecer primero el tipo de dato que se introducirá y utilizar un **switch-case** para llamar al constructor adecuado en cada caso.
  - Fíjese que de esta forma los elementos del arreglo son punteros a clase base **Figure**, por ello pueden apuntar a objetos de cualquiera de sus clases derivadas.

Problema a resolver:

Si no se usa el polimorfismo:

```
//En el programa principal, luego de la entrada de datos:
for(int i=0;i<20;i++)
  {
    switch(Array[i]->getTipo())
    {
      case 1:((Circle*)Array[i])->print();
            cout<<((Circle*)Array[i])->area()<<endl;
            break;
      case 2:((Rectangle*)Array[i])->print();
            cout<<((Rectangle*)Array[i])->area()<<endl;
            break;
      case 3:((Pie*)Array[i])->print();
            cout<<((Pie*)Array[i])->area()<<endl;
    }
  }
```

```
//Antes de terminar el programa se deben destruir los
//objetos creados dinámicamente:

for(int i=0;i<20;i++)
  {
    switch(Array[i]->getTipo())
    {
      case 1: delete (Circle*)Array[i];break;
      case 2: delete (Rectangle*)Array[i];break;
      case 3: delete (Pie*)Array[i];
    }
  }
  return 0;
}
//Fin del programa principal (main())
```

➢ *Problema a resolver:*

- Observaciones:

  - Fíjese que en este caso para poder llamar a los métodos adecuados **print()** y **area()** primero se ha de preguntar por su atributo **tipo** y según lo obtenido, realizar el casteo al objeto adecuado.
  - Si no se hiciera el casteo siempre llamaría al **print()** y **area()** de **Figure**.
  - Cada nuevo objeto que es apuntado por cada elemento del arreglo fue creado dinámicamente por lo que se debe llamar al operador **delete** con cada elemento del arreglo para destruirlos.
  - Como cada elemento del arreglo es un puntero a **Figure** el operador **delete** solo llama al destructor de esta clase base y no al destructor de un **Circle**, un **Rectangle** o un **Pie** lo cual conlleva a un error. Por ello nuevamente se realiza el casteo, o sea, para llamar al destructor de la clase derivada adecuada.

➢ *Problema a resolver:*

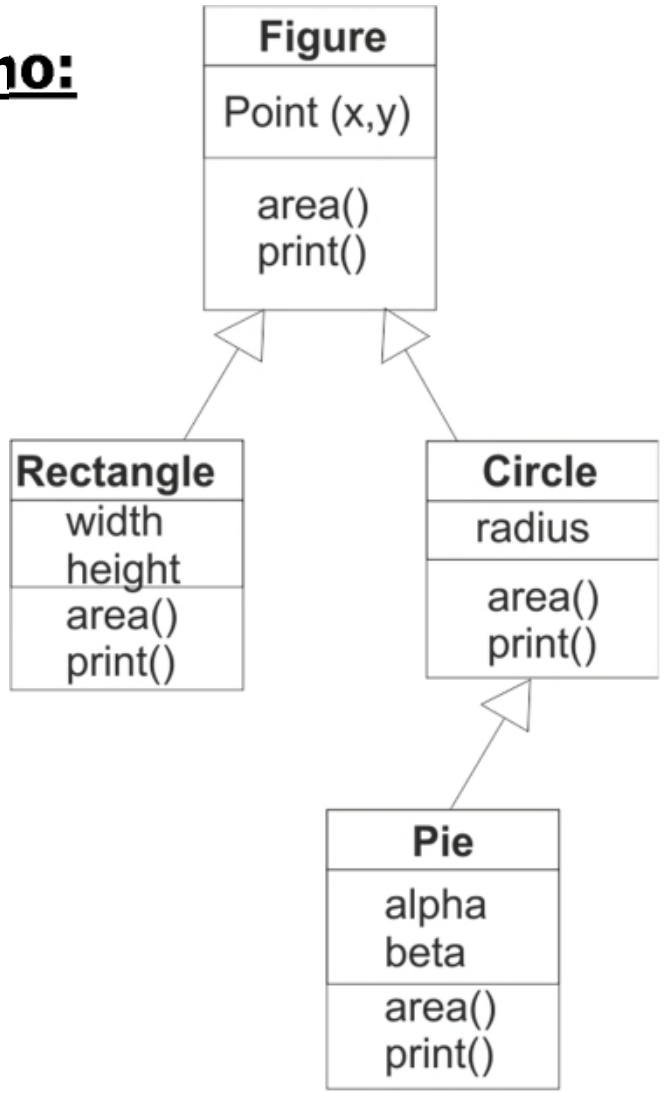
▯ Si se usa el polimorfismo:

```
//En el programa principal, luego de la entrada
//de datos:

   for(int i=0;i<20;i++)
   {
      Array[i]->print();
      cout<<Array[i]->area()<<endl;
   }
```

```
//Antes de terminar el programa se deben destruir
//los objetos creados dinámicamente:

   for(int i=0;i<20;i++)
   {
      delete Array[i];
   }
   return 0;
}
//Fin del programa principal (main())
```

- Al utilizar el polimorfismo se logra determinar en tiempo de ejecución el método **print()** y **area()** adecuado según el objeto al que pertenece, y llamar a éste.
- Igualmente se logra determinar el destructor correcto a ser llamado.

- ***Problema a resolver:***
  - **Si se usa el polimorfismo:**



Polimorfismo y funciones virtuales.

- Lo anterior es posible lograrlo si utilizamos las ***funciones virtuales*** para lograr el ***polimorfismo.*** Primero debemos seleccionar las funciones que son aplicables tanto a las clases base como a las clases derivadas (area, print). Luego a esa función se le coloca en su prototipo la palabra reservada **virtual**

```
class Figure{
public:
  …
  virtual float area() const;
  virtual void print() const;
  …
};
```

Polimorfismo y funciones virtuales.

Al declarar las funciones virtuales:

- El compilador generar un mecanismo mediante el cual, en tiempo de ejecución determina cual es la función correcta que tiene que invocar, independientemente de que la referencia sea declarada a la clase base.

- Este mecanismo determina a qué función concreta tiene que llamar averiguando a qué objeto de la jerarquía está apuntando en ese momento el puntero de la clase base.

Polimorfismo y funciones virtuales.

- Observaciones:

  - Si una función se declara virtual en una clase base cualquiera, a partir de ese momento todas las clases derivadas de ellas que redefinan esa función, tendrán esa función como virtual.
  - A este comportamiento se le denomina ***polimorfismo*** o ligadura dinámica (o ligadura tardía).
  - Solo es posible lograrlo a través de punteros a la clase base. Cuando son objetos y no punteros a clase base los métodos se llaman de la forma tradicional (ligadura estática).
  - Implican una sobrecarga de función, por lo cual debemos definir como virtuales solo aquellos métodos que utilizaremos de forma ***polimórfica***.
  - Los constructores no pueden ser declarados como funciones virtuales.

Polimorfismo y Destructores virtuales.

- En el caso de no usar polimorfismo para poder llamar al destructor correcto del objeto y no al de su clase base se tenía que castear adecuadamente.
- Usando el polimorfismo ese problema se puede resolver declarando al destructor de la clase base como virtual. Esto hará que cada uno de los destructores de las clases derivadas sea también virtual y por lo tanto se llame al destructor de la clase apropiada.

```
class Figure{
public:
  ...
  virtual ~Figure(){}
  ...
};
```

➢ Clases abstractas y métodos virtuales puros.

- **¿Tiene sentido la existencia de una clase con la cual no se puedan crear objetos?**

- Sí, cuando la clase se usará sólo como base para la confección de otras clases derivadas, y esta clase no representa a un objeto concreto a modelar. Esta sería una **clase abstracta**. En nuestro caso sería la clase **Figure** que no es ninguna figura específica de la realidad, y se utiliza para la confección de **Circle**, **Rectangle** e indirectamente de **Pie**.

➢ Clases abstractas y métodos virtuales puros.

- **¿De qué forma se podría evitar que se puedan crear objetos de una clase?**

- Poniendo el constructor privado o protegido sería una forma, pero no es la forma utilizada en el polimorfismo.
- Para ello lo que se hace es definir un **método virtual puro,** o sea, en la definición del prototipo de un método virtual se le adiciona al final **=0**. Estos métodos de la clase base abstracta son los que no tiene sentido implementar mientras no se refiera a un objeto concreto. En este caso se podría poner como ejemplo al método **area()** de **Figure**.

Clases abstractas y métodos virtuales puros.

- **Ejemplo:**

```
class Figure{
public:
  ...
  virtual float area() const=0;
  virtual void print() const;
  ...
};
```

Nota: **print()** no se iguala a cero porque su implementación se utiliza en las clases derivadas.

POLIMORFISMO EN POO

- Observaciones:

  - Los métodos virtuales puros definen a la clase como una clase abstracta.
  - Los métodos virtuales puros no tienen implementación.
  - No se pueden crear objetos de clases abstracta. (tampoco es conceptualmente correcto)
  - Lo que se puede hacer es crear punteros a clases base abstracta.
  - Si una clase derivada de una clase abstracta no redefine el método virtual puro, esta clase se considera abstracta y tampoco se pueden crear objetos de ella.

Ejercicio integrador para autoestudio

- En una cafetería existe una máquina dispensadora de bebidas que dispensa café, café con leche, y chocolate. El cliente puede solicitar el producto caliente, tibio o al tiempo. Luego de realizar la orden, la máquina calcula su precio y lo visualiza en la pantalla, el cliente introduce el dinero, y si este es igual o mayor que el precio visualizado la máquina devuelve la diferencia, y prepara y entrega la bebida. A su vez registra el producto dispensado en la memoria de la máquina para que el administrador pueda calcular el gasto de cada ingrediente, y el dinero recaudado.

Bebida	Ingrediente	Volumen que consume (variable en este rango por el cliente)	Precio
Café	Café en polvo	30-50ml	1.00 pesos
	Agua	70-100ml	
Café con leche	Café en polvo	30-50ml	2.00 pesos
	Agua	50-70ml	
	Leche	60-80ml	
Chocolate	Chocolate	60-100ml	2.50 pesos
	Agua	130-170ml	

- Bebida caliente: +0.20 pesos, bebida tibia: +0.10 pesos.

CONCLUSIONES