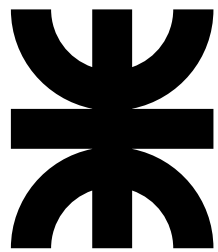
# Informática II

## Funciones virtuales y polimorfismo en C++

Gonzalo F. Pérez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2018 –

# Introducción

- Las funciones virtuales y el polimorfismo permiten diseñar programas escalables
- Los programas puede tratar a los objetos de una jerarquía de clases de forma genérica como objetos de la clase base

# Introducción

- Las funciones virtuales y el polimorfismo permiten diseñar programas escalables
- Los programas puede tratar a los objetos de una jerarquía de clases de forma genérica como objetos de la clase base

- Otra forma de manejar objetos diferentes de forma general es mediante una estructura de selección `switch`, que realice acciones adecuadas sobre cada objeto en particular a partir del tipo de objeto

# Introducción

- Las funciones virtuales y el polimorfismo permiten diseñar programas escalables
- Los programas puede tratar a los objetos de una jerarquía de clases de forma genérica como objetos de la clase base

- Otra forma de manejar objetos diferentes de forma general es mediante una estructura de selección `switch`, que realice acciones adecuadas sobre cada objeto en particular a partir del tipo de objeto
- Por ejemplo: cálculo del área de figuras geométricas como cuadrado o círculo heredadas a partir de una clase figura

# Introducción

- Las funciones virtuales y el polimorfismo permiten diseñar programas escalables
- Los programas puede tratar a los objetos de una jerarquía de clases de forma genérica como objetos de la clase base

- Otra forma de manejar objetos diferentes de forma general es mediante una estructura de selección `switch`, que realice acciones adecuadas sobre cada objeto en particular a partir del tipo de objeto
- Por ejemplo: cálculo del área de figuras geométricas como cuadrado o círculo heredadas a partir de una clase figura

Inconvenientes con el uso de `switch`

1. Olvidar una evaluación de algún tipo particular de objeto

# Introducción

- Las funciones virtuales y el polimorfismo permiten diseñar programas escalables
- Los programas puede tratar a los objetos de una jerarquía de clases de forma genérica como objetos de la clase base

- Otra forma de manejar objetos diferentes de forma general es mediante una estructura de selección `switch`, que realice acciones adecuadas sobre cada objeto en particular a partir del tipo de objeto
- Por ejemplo: cálculo del área de figuras geométricas como cuadrado o círculo heredadas a partir de una clase figura

Inconvenientes con el uso de `switch`

1. Olvidar una evaluación de algún tipo particular de objeto
2. Obvidar evaluar todos los casos posibles del `switch`

# Introducción

- Las funciones virtuales y el polimorfismo permiten diseñar programas escalables
- Los programas puede tratar a los objetos de una jerarquía de clases de forma genérica como objetos de la clase base

- Otra forma de manejar objetos diferentes de forma general es mediante una estructura de selección `switch`, que realice acciones adecuadas sobre cada objeto en particular a partir del tipo de objeto
- Por ejemplo: cálculo del área de figuras geométricas como cuadrado o círculo heredadas a partir de una clase figura

Inconvenientes con el uso de `switch`

1. Olvidar una evaluación de algún tipo particular de objeto
2. Obvidar evaluar todos los casos posibles del `switch`
3. Si se agregan nuevos tipos hay que agregar este nuevo caso a la estructura `switch`

# Funciones virtuales

- Supongamos un conjunto de clases como `Circulo`, `Triangulo`, `Rectangulo`, `Cuadrado`, etc. todas derivadas de una clase `Figura`

# Funciones virtuales

- Supongamos un conjunto de clases como `Circulo`, `Triangulo`, `Rectangulo`, `Cuadrado`, etc. todas derivadas de una clase `Figura`
- En un programa POO cada objeto de `Figura` podría tener la capacidad de dibujarse en pantalla

# Funciones virtuales

- Supongamos un conjunto de clases como `Circulo`, `Triangulo`, `Rectangulo`, `Cuadrado`, etc. todas derivadas de una clase `Figura`
- En un programa POO cada objeto de `Figura` podría tener la capacidad de dibujarse en pantalla
- La función `dibujar` de cada clase debe ser diferente

# Funciones virtuales

- Supongamos un conjunto de clases como `Circulo`, `Triangulo`, `Rectangulo`, `Cuadrado`, etc. todas derivadas de una clase `Figura`
- En un programa POO cada objeto de `Figura` podría tener la capacidad de dibujarse en pantalla
- La función `dibujar` de cada clase debe ser diferente
- Aunque sería interesante poder tratar a todas las figuras de manera genérica como objeto de la clase `Figura`

# Funciones virtuales

- Supongamos un conjunto de clases como `Circulo`, `Triangulo`, `Rectangulo`, `Cuadrado`, etc. todas derivadas de una clase `Figura`
- En un programa POO cada objeto de `Figura` podría tener la capacidad de dibujarse en pantalla
- La función `dibujar` de cada clase debe ser diferente
- Aunque sería interesante poder tratar a todas las figuras de manera genérica como objeto de la clase `Figura`
- De esta forma, para dibujar cualquier figura particular se podría llamar directamente a la función miembro `dibujar` de la clase base (`Figura`)

# Funciones virtuales

- Supongamos un conjunto de clases como `Circulo`, `Triangulo`, `Rectangulo`, `Cuadrado`, etc. todas derivadas de una clase `Figura`
- En un programa POO cada objeto de `Figura` podría tener la capacidad de dibujarse en pantalla
- La función `dibujar` de cada clase debe ser diferente
- Aunque sería interesante poder tratar a todas las figuras de manera genérica como objeto de la clase `Figura`
- De esta forma, para dibujar cualquier figura particular se podría llamar directamente a la función miembro `dibujar` de la clase base (`Figura`)
- En este caso el programa debe determinar en tiempo de ejecución, de forma dinámica, qué función `dibujar` de las clases derivada debe ejecutar

# Funciones virtuales

- Supongamos un conjunto de clases como `Circulo`, `Triangulo`, `Rectangulo`, `Cuadrado`, etc. todas derivadas de una clase `Figura`
- En un programa POO cada objeto de `Figura` podría tener la capacidad de dibujarse en pantalla
- La función `dibujar` de cada clase debe ser diferente
- Aunque sería interesante poder tratar a todas las figuras de manera genérica como objeto de la clase `Figura`
- De esta forma, para dibujar cualquier figura particular se podría llamar directamente a la función miembro `dibujar` de la clase base (`Figura`)
- En este caso el programa debe determinar en tiempo de ejecución, de forma dinámica, qué función `dibujar` de las clases derivada debe ejecutar

Se puede lograr este comportamiento si la función `dibujar` se declara en la clase base como *función virtual*

# Funciones virtuales

- Supongamos un conjunto de clases como **`Circulo`**, **`Triangulo`**, **`Rectangulo`**, **`Cuadrado`**, etc. todas derivadas de una clase **`Figura`**
- En un programa POO cada objeto de **`Figura`** podría tener la capacidad de dibujarse en pantalla
- La función **`dibujar`** de cada clase debe ser diferente
- Aunque sería interesante poder tratar a todas las figuras de manera genérica como objeto de la clase **`Figura`**
- De esta forma, para dibujar cualquier figura particular se podría llamar directamente a la función miembro **`dibujar`** de la clase base (**`Figura`**)
- En este caso el programa debe determinar en tiempo de ejecución, de forma dinámica, qué función **`dibujar`** de las clases derivada debe ejecutar

Se puede lograr este comportamiento si la función **`dibujar`** se declara en la clase base como *función virtual*

Una función se declara como virtual mediante la palabra reservada **`virtual`**, como

```
virtual void dibujar() const;
```

# Funciones virtuales

```
class Figura {
  virtual void dibujar() const;
  . . .
};
```

# Funciones virtuales

```
class Figura {
  virtual void dibujar() const;
  . . .
};

class Cuadrado : public Figura {
  . . .
};
```

# Funciones virtuales

```
class Figura {
  virtual void dibujar() const;
  . . .
};

class Cuadrado : public Figura {
  . . .
};

class Circulo : public Figura {
  . . .
};
```

# Funciones virtuales

```
class Figura {
  virtual void dibujar() const;
  . . .
};

class Cuadrado : public Figura {
  . . .
};

class Circulo : public Figura {
  . . .
};
```

```
1  int main() {
2    Cuadrado cuad;
3    Circulo circ;
4
5    Figura *ptrFig[2];
6    ptrFig[0] = &cuad;
7    ptrFig[1] = &circ;
8
9    for(int i = 0; i < 2; i++)
10     (ptrFig[i])->dibujar();
11
12   return 0;
13 }
```

# Funciones virtuales

- Supongamos que la función miembro `dibujar` se declara como `virtual` en la clase base

# Funciones virtuales

- Supongamos que la función miembro `dibujar` se declara como `virtual` en la clase base
- Y luego se utiliza un puntero o una referencia de la clase base para apuntar/referenciar un objeto de una clase derivada

# Funciones virtuales

- Supongamos que la función miembro `dibujar` se declara como `virtual` en la clase base
- Y luego se utiliza un puntero o una referencia de la clase base para apuntar/referenciar un objeto de una clase derivada
- Cuando se invoca a la función `dibujar` (p.e. haciendo `ptrFigura->dibujar()`) el programa elige de forma dinámica (en tiempo de ejecución) a la función `dibujar` adecuada de la clase derivada, utilizando para ello el tipo de objeto particular y no el tipo de puntero o referencia

$$\longrightarrow \textit{vinculación dinámica}$$

# Funciones virtuales

- Supongamos que la función miembro `dibujar` se declara como `virtual` en la clase base
- Y luego se utiliza un puntero o una referencia de la clase base para apuntar/referenciar un objeto de una clase derivada
- Cuando se invoca a la función `dibujar` (p.e. haciendo `ptrFigura->dibujar()`) el programa elige de forma dinámica (en tiempo de ejecución) a la función `dibujar` adecuada de la clase derivada, utilizando para ello el tipo de objeto particular y no el tipo de puntero o referencia

$$\longrightarrow \textit{vinculación dinámica}$$

- Si la función virtual se invoca haciendo referencia a un objeto específico (p.e. `objetoCuadrado.dibujar()`) la referencia se resuelve en tiempo de compilación

$$\longrightarrow \textit{vinculación estática}$$

# Clase base abstracta

- Si se piensa en una clase como un tipo de dato es correcto asumir que se generarán instancias de ese tipo de datos, o sea, objetos

# Clase base abstracta

- Si se piensa en una clase como un tipo de dato es correcto asumir que se generarán instancias de ese tipo de datos, o sea, objetos
- Existen algunos casos donde resulta útil definir clases de las que nunca se instanciarán objetos $\longrightarrow$ *clases abstractas*

# Clase base abstracta

- Si se piensa en una clase como un tipo de dato es correcto asumir que se generarán instancias de ese tipo de datos, o sea, objetos
- Existen algunos casos donde resulta útil definir clases de las que nunca se instanciarán objetos $\longrightarrow$ *clases abstractas*

- Las *clases bases abstractas* se utilizan para heredar otras clases llamadas *clases concretas* (las que permiten instanciar objetos)

# Clase base abstracta

- Si se piensa en una clase como un tipo de dato es correcto asumir que se generarán instancias de ese tipo de datos, o sea, objetos
- Existen algunos casos donde resulta útil definir clases de las que nunca se instanciarán objetos $\longrightarrow$ *clases abstractas*

- Las *clases bases abstractas* se utilizan para heredar otras clases llamadas *clases concretas* (las que permiten instanciar objetos)
- Las clases bases abstractas resultan demasiado genéricas como para definir objetos reales

# Clase base abstracta

- Si se piensa en una clase como un tipo de dato es correcto asumir que se generarán instancias de ese tipo de datos, o sea, objetos
- Existen algunos casos donde resulta útil definir clases de las que nunca se instanciarán objetos $\longrightarrow$ *clases abstractas*

- Las *clases bases abstractas* se utilizan para heredar otras clases llamadas *clases concretas* (las que permiten instanciar objetos)
- Las clases bases abstractas resultan demasiado genéricas como para definir objetos reales
- Una clase resulta abstracta cuando tiene una o más funciones virtuales "puras"

# Clase base abstracta

- Si se piensa en una clase como un tipo de dato es correcto asumir que se generarán instancias de ese tipo de datos, o sea, objetos
- Existen algunos casos donde resulta útil definir clases de las que nunca se instanciarán objetos $\longrightarrow$ *clases abstractas*

- Las *clases bases abstractas* se utilizan para heredar otras clases llamadas *clases concretas* (las que permiten instanciar objetos)
- Las clases bases abstractas resultan demasiado genéricas como para definir objetos reales
- Una clase resulta abstracta cuando tiene una o más funciones virtuales "puras"
- Una *función virtual pura* es aquella que tienen un inicializador igual a cero en su declaración

# Clase base abstracta

- Si se piensa en una clase como un tipo de dato es correcto asumir que se generarán instancias de ese tipo de datos, o sea, objetos
- Existen algunos casos donde resulta útil definir clases de las que nunca se instanciarán objetos $\longrightarrow$ *clases abstractas*

- Las *clases bases abstractas* se utilizan para heredar otras clases llamadas *clases concretas* (las que permiten instanciar objetos)
- Las clases bases abstractas resultan demasiado genéricas como para definir objetos reales
- Una clase resulta abstracta cuando tiene una o más funciones virtuales "puras"
- Una *función virtual pura* es aquella que tienen un inicializador igual a cero en su declaración

```
virtual void dibujar() const = 0; // función virtual pura
```

# Clase base abstracta

- Si se piensa en una clase como un tipo de dato es correcto asumir que se generarán instancias de ese tipo de datos, o sea, objetos
- Existen algunos casos donde resulta útil definir clases de las que nunca se instanciarán objetos $\longrightarrow$ *clases abstractas*

- Las *clases bases abstractas* se utilizan para heredar otras clases llamadas *clases concretas* (las que permiten instanciar objetos)
- Las clases bases abstractas resultan demasiado genéricas como para definir objetos reales
- Una clase resulta abstracta cuando tiene una o más funciones virtuales "puras"
- Una *función virtual pura* es aquella que tienen un inicializador igual a cero en su declaración

```
virtual void dibujar() const = 0; // función virtual pura
```

Ver ejemplo de la clase abstracta `Figura` y sus clases concretas derivadas `Cuadrado` y `Circulo`

# Polimorfismo

- Permite que objetos diferentes relacionados mediante herencia respondan de manera diferentes al mismo mensaje, o sea a una llamada de función miembro

# Polimorfismo

- Permite que objetos diferentes relacionados mediante herencia respondan de manera diferentes al mismo mensaje, o sea a una llamada de función miembro
- El mismo mensaje enviado a objetos de diferentes tipos toman “diferentes formas”, de ahí el término *polimorfismo*

# Polimorfismo

- Permite que objetos diferentes relacionados mediante herencia respondan de manera diferentes al mismo mensaje, o sea a una llamada de función miembro
- El mismo mensaje enviado a objetos de diferentes tipos toman “diferentes formas”, de ahí el término *polimorfismo*
- El polimorfiso se implementa mediante funciones virtuales

# Polimorfismo – Ejemplo

```
1 Empleado e, *ptrE = &e; // Clase base 'Empleado'
2
3
4
5
6
```

# Polimorfismo – Ejemplo

```
1 Empleado e, *ptrE = &e; // Clase base 'Empleado'
2 EmpleadoXHora h, *ptrH = &h; // Clase derivada 'EmpleadoXHora'
3
4
5
6
```

# Polimorfismo – Ejemplo

```
1 Empleado e, *ptrE = &e; // Clase base 'Empleado'
2 EmpleadoXHora h, *ptrH = &h; // Clase derivada 'EmpleadoXHora'
3
4
5
6
```

- La clase base y la clase derivada tienen sus propias funciones `imprime` definidas

# Polimorfismo – Ejemplo

```
1 Empleado e, *ptrE = &e; // Clase base 'Empleado'
2 EmpleadoXHora h, *ptrH = &h; // Clase derivada 'EmpleadoXHora'
3 ptrE->imprime(); // llama a 'imprime' de la clase base
4
5
6
```

- La clase base y la clase derivada tienen sus propias funciones `imprime` definidas
- Las funciones `imprime` no están declaradas como virtuales y tienen la misma firma, por lo que invocarla mediante un puntero `Empleado` da como resultado una llamada a `Empleado::imprime()`

# Polimorfismo – Ejemplo

```
1 Empleado e, *ptrE = &e; // Clase base 'Empleado'
2 EmpleadoXHora h, *ptrH = &h; // Clase derivada 'EmpleadoXHora'
3 ptrE->imprime(); // llama a 'imprime' de la clase base
4
5
6
```

- La clase base y la clase derivada tienen sus propias funciones **`imprime`** definidas
- Las funciones **`imprime`** no están declaradas como virtuales y tienen la misma firma, por lo que invocarla mediante un puntero **`Empleado`** da como resultado una llamada a **`Empleado::imprime()`**
- Indipendientemente de si el puntero **`Empleado`** apunta a un objeto de la clase base o de la clase devivada

# Polimorfismo – Ejemplo

```
1 Empleado e, *ptrE = &e; // Clase base 'Empleado'
2 EmpleadoXHora h, *ptrH = &h; // Clase derivada 'EmpleadoXHora'
3 ptrE->imprime(); // llama a 'imprime' de la clase base
4 ptrH->imprime(); // llama a 'imprime' de la clase derivada
5
6
```

- La clase base y la clase derivada tienen sus propias funciones `imprime` definidas
- Las funciones `imprime` no están declaradas como virtuales y tienen la misma firma, por lo que invocarla mediante un puntero `Empleado` da como resultado una llamada a `Empleado::imprime()`
- Indipendientemente de si el puntero `Empleado` apunta a un objeto de la clase base o de la clase devivada
- Invocar a la función `imprime` usando un puntero a `EmpleadoXHora` llama a la función `EmpleadoXHora::imprime()`

# Polimorfismo – Ejemplo

```
1 Empleado e, *ptrE = &e; // Clase base 'Empleado'
2 EmpleadoXHora h, *ptrH = &h; // Clase derivada 'EmpleadoXHora'
3 ptrE->imprime(); // llama a 'imprime' de la clase base
4 ptrH->imprime(); // llama a 'imprime' de la clase derivada
5 ptrE = &h; // conversión implícita permisible
6
```

- La clase base y la clase derivada tienen sus propias funciones `imprime` definidas
- Las funciones `imprime` no están declaradas como virtuales y tienen la misma firma, por lo que invocarla mediante un puntero `Empleado` da como resultado una llamada a `Empleado::imprime()`
- Indipendientemente de si el puntero `Empleado` apunta a un objeto de la clase base o de la clase devivada
- Invocar a la función `imprime` usando un puntero a `EmpleadoXHora` llama a la función `EmpleadoXHora::imprime()`
- Un objeto de la clase `EmpleadoXHora` es también un objeto de la clase `Empleado`

# Polimorfismo – Ejemplo

```
1 Empleado e, *ptrE = &e; // Clase base 'Empleado'
2 EmpleadoXHora h, *ptrH = &h; // Clase derivada 'EmpleadoXHora'
3 ptrE->imprime(); // llama a 'imprime' de la clase base
4 ptrH->imprime(); // llama a 'imprime' de la clase derivada
5 ptrE = &h; // conversión implícita permisible
6 ptrE->imprime(); // aún llama a 'imprime' de la clase base
```

- La clase base y la clase derivada tienen sus propias funciones **`imprime`** definidas
- Las funciones **`imprime`** no están declaradas como virtuales y tienen la misma firma, por lo que invocarla mediante un puntero **`Empleado`** da como resultado una llamada a **`Empleado::imprime()`**
- Indipendientemente de si el puntero **`Empleado`** apunta a un objeto de la clase base o de la clase devivada
- Invocar a la función **`imprime`** usando un puntero a **`EmpleadoXHora`** llama a la función **`EmpleadoXHora::imprime()`**
- Un objeto de la clase **`EmpleadoXHora`** es también un objeto de la clase **`Empleado`**

# Polimorfismo – Ejemplo

```
1 Empleado e, *ptrE = &e; // Clase base 'Empleado'
2 EmpleadoXHora h, *ptrH = &h; // Clase derivada 'EmpleadoXHora'
3 ptrE->imprime(); // llama a 'imprime' de la clase base
4 ptrH->imprime(); // llama a 'imprime' de la clase derivada
5 ptrE = &h; // conversión implícita permisible
6 ptrE->imprime(); // aún llama a 'imprime' de la clase base
```

- La clase base y la clase derivada tienen sus propias funciones **`imprime`** definidas
- Las funciones **`imprime`** no están declaradas como virtuales y tienen la misma firma, por lo que invocarla mediante un puntero **`Empleado`** da como resultado una llamada a **`Empleado::imprime()`**
- Indipendientemente de si el puntero **`Empleado`** apunta a un objeto de la clase base o de la clase devivada
- Invocar a la función **`imprime`** usando un puntero a **`EmpleadoXHora`** llama a la función **`EmpleadoXHora::imprime()`**
- Un objeto de la clase **`EmpleadoXHora`** es también un objeto de la clase **`Empleado`**
- Para llamar a **`imprime`** de la clase base mediante un puntero a un objeto de la clase derivada hay que hacer **`ptrH->Empleado::imprime()`**

# Ejemplo: clase abstracta `Figura` y derivadas `Punto`, `Circulo` y `Cilindro`

```
class Figura { // Clase abstracta
  public:
    virtual double area() const { return 0.0; }
    virtual double volumen() const { return 0.0; }

    // funciones virtuales puras sustituidas en clases derivadas
    virtual void imprimeNombreFigura() const = 0;
    virtual void imprime() const = 0;
};
```

# Ejemplo: clase abstracta `Figura` y derivadas `Punto`, `Circulo` y `Cilindro`

```
class Figura { // Clase abstracta
  public:
    virtual double area() const { return 0.0; }
    virtual double volumen() const { return 0.0; }

    // funciones virtuales puras sustituidas en clases derivadas
    virtual void imprimeNombreFigura() const = 0;
    virtual void imprime() const = 0;
};
```

```
class Punto : public Figura {
  // Define 'imprimeNombreFigura' e 'imprime'
};
```

# Ejemplo: clase abstracta `Figura` y derivadas `Punto`, `Circulo` y `Cilindro`

```
class Figura { // Clase abstracta
  public:
    virtual double area() const { return 0.0; }
    virtual double volumen() const { return 0.0; }

    // funciones virtuales puras sustituidas en clases derivadas
    virtual void imprimeNombreFigura() const = 0;
    virtual void imprime() const = 0;
};
```

```
class Punto : public Figura {
  // Define 'imprimeNombreFigura' e 'imprime'
};
```

```
class Circulo : public Punto {
  // Define 'imprimeNombreFigura', 'imprime' y 'area'
};
```

# Ejemplo: clase abstracta `Figura` y derivadas `Punto`, `Circulo` y `Cilindro`

```
class Figura { // Clase abstracta
  public:
    virtual double area() const { return 0.0; }
    virtual double volumen() const { return 0.0; }

    // funciones virtuales puras sustituidas en clases derivadas
    virtual void imprimeNombreFigura() const = 0;
    virtual void imprime() const = 0;
};
```

```
class Punto : public Figura {
  // Define 'imprimeNombreFigura' e 'imprime'
};
```

```
class Circulo : public Punto {
  // Define 'imprimeNombreFigura', 'imprime' y 'area'
};
```

```
class Cilindro : public Circulo {
  // Define 'imprimeNombreFigura', 'imprime', 'area' y 'volumen'
};
```

# Ejemplo: clase abstracta `Figura` y derivadas `Punto`, `Circulo` y `Cilindro`

```
class Figura { // Clase abstracta
  public:
    virtual double area() const { return 0.0; }
    virtual double volumen() const { return 0.0; }

    // funciones virtuales puras sustituidas en clases derivadas
    virtual void imprimeNombreFigura() const = 0;
    virtual void imprime() const = 0;
};
```

```
class Punto : public Figura {
  // Define 'imprimeNombreFigura' e 'imprime'
};
```

```
class Circulo : public Punto {
  // Define 'imprimeNombreFigura', 'imprime' y 'area'
};
```

```
class Cilindro : public Circulo {
  // Define 'imprimeNombreFigura', 'imprime', 'area' y 'volumen'
};
```

Ver código fuente `fig20_01.cpp` (`punto1`, `circulo1`, `cilindro1`)