



Universidad
Carlos III de Madrid

Práctica de Informática Industrial I

Grado en Ingeniería en Electrónica Industrial y Automática

2014-2015

Práctica 4

Herencia

Profesores:

Fares Abu-Dakka

Mohamed Abderrahim

Álvaro Castro González

Avinash Ranganath

Carlos Alonso

Ioannis Douratsos

Práctica 4 – Herencia.

La herencia es una funcionalidad de los lenguajes objetos que permite reutilizar el código de clases existentes y extenderlo.

1. Herencia para extender

En C++ se puede declarar una clase (llamada clase hija) que derive de otra clase (clase padre). Esto implica que la clase hija recuperará todos los miembros de la clase padre y puede ampliarse con nuevas funciones miembro.

El siguiente ejemplo muestra cómo se puede definir una clase punto con color (el color se representará con un entero) derivada de la clase `Point` para recuperar la gestión de las coordenadas que ya se ha implementado.

Poniendo el siguiente código en el `.h` de la clase `Point`:

```
class Point {
public:
    Point(int x, int y);

    void set(int x, int y);
    int getX() const {return _x;}
    int getY() const {return _y;}

    void display () const;

private:
    int _x, _y;
}
```

Y el siguiente en el `.h` de la nueva clase `ColorPoint`:

```
class ColorPoint : public Point {
public:
    ColorPoint(int x, int y, int c) : Point(x, y) {_color = c;}
    int getColor() const {return _color;}

private:
    int _color;
}
```

Para el siguiente código de prueba:

```
int main() {
    ColorPoint CP(10, 20, 255);

    CP.getX();           // come from Point class
    CP.display();        // come from Point class, display x and y
    CP.getColor();       // come from PointColor class
}
```

Gracias a la herencia, la clase `ColorPoint` sólo tiene que definir su miembro adicional (`getColor`), y recuperar de la clase `Point` los miembros relacionados con las coordenadas. Notad como el constructor de `PointColor` llama al constructor de `Point` que tomará dos parámetros.

2. Redefinición de una función miembro

Esta solución todavía no es completamente satisfactoria ya que la función `display`, heredada de `Point` no muestra el color del punto. Por ello se puede redefinir las funciones miembros cuyo comportamiento queremos cambiar en la clase hija.

Así, la clase `Point` quedará:

```
class Point {
public:
    Point(int x, int y);
    // set, getX, getY
    void display() const {cout << _x << _y;}

private:
    int _x, _y;
};
```

Y la clase `ColorPoint`:

```
class ColorPoint : public Point {
public:
    ColorPoint(int x, int y, int c) : Point(x, y) {_color = c;}
    int getColor() const {return _color;}
    void display() const {cout << _x << _y << _color;}

private:
    int _color;
};
```

De este modo, se puede cambiar el comportamiento de una clase hija con la redefinición de sus propios métodos.

Con lo que el `main` quedaría:

```
int main() {
    ColorPoint CP(10, 20, 255);
    CP.display();    // come from PointColor class, display x, y and color

    Point P(10, 20);
    P.display();    // come from Point class, display x and y
}
```

Nota: En este momento la clase ColorPoint no compilaría. Encontrar el error y proponer una solución.

Ejercicios

- Realizar un simulador de ordenador:
 - o Para ilustrar los conceptos de la programación orientada a objetos en C++, se va a realizar un pequeño proyecto a lo largo de las prácticas. Este se basará en desarrollar un simulador de ordenador. El programa deberá simular teclados, procesadores, programas, pantallas, etc. Para que el programa quede sencillo, nuestro ordenador solo será capaz de manejar cadenas de caracteres. De momento se propone implementar el código que corresponde al diagrama UML del anexo.
 - o En concreto, el programa deberá poder leer caracteres desde el teclado, transmitirlos al procesador para que los transforme y enviarlos a una pantalla que muestre la cadena procesada. El diseño deberá estar pensado para que sea modular, es decir, que sea relativamente sencillo añadir o modificar cualquier elemento de las clases.
 - o Para cada clase se debe crear un fichero de cabecera `nombre_clase.h` y un fichero fuente `nombre_clase.cpp`.
 - o La descripción de los métodos a implementar se puede encontrar en la tabla del anexo.

Anexo

Clase	Métodos	Descripción
Device	Device::Device(const string & name)	Inicializa un componente con el nombre especificado
	const string & Device::getName() const	Devuelve el nombre del componente.
Display	Display::Display(const string & name)	Inicializa una pantalla con el nombre especificado.
	void Display::process(const string & data)	Muestra la cadena por pantalla utilizando cout.
Processor	Processor::Processor(const string & name)	Inicializa un procesador con el nombre especificado.
	void Processor::connectTo(Display & display)	Conecta el procesador con un Display.
	void Processor::process(const string & data)	Invierte la cadena de caracteres data y añade el prefijo “PROCESSED”. Manda el resultado al Display al cual está conectado. Por ejemplo si data vale “hola”, la cadena enviada al Display debe ser “PROCESSED: aloh”.
Keyboard	Keyboard::Keyboard(const string & name)	Inicializa un teclado con el nombre especificado.
	void Keyboard::connectTo(Processor & cpu)	Conecta el teclado con un procesador.
	void Keyboard::process()	Implementación por defecto de un teclado, lee una cadena de caracteres desde el teclado real utilizando el operador de flujo cin. Después manda esta cadena al procesador llamando al método process() del procesador al cual está conectado.
CharKeyboard	CharKeyboard::CharKeyboard(const string & name)	Inicializa un teclado de carácter con el nombre especificado.
	void CharKeyboard::process()	Igual que Keyboard::process() pero solo lee un carácter (para ello se puede utilizar la función estándar cin.get()).
LineKeyboard	LineKeyboard::LineKeyboard(const string & name)	Inicializa un teclado de línea con el nombre especificado.
	void LineKeyboard::process()	Igual que Keyboard::process() pero lee una línea completa, es decir, lee caracteres hasta encontrar un '\n' y manda toda la línea directamente al procesador.

