FACULTAD DE INFORMÁTICA

Métodos virtuales, polimorfismo y clases abstractas

TEMA

Programación orientada a objetos — Unidad 9

Autor: Luis Hernández Yáñez

FdI UCM

Contenido

FdI UCM

Listas de objetos dinámicos

```
// Clase Lista - Archivo de cabecera "listaper.h"
#ifndef listaper_h
#define listaper_h
#include "persona0.h" // Clase Persona de la unidad anterior

class Lista {
public:
  Lista();
  ~Lista();
  bool llena() const;
  bool vacia() const;
  bool insertar(Persona*);
  bool recuperar(int, Persona*&) const;
  void mostrar() const;
private:
  enum { MAX = 100 };
  Persona* _array[MAX]; // Punteros a Personas
  int _cont;
};

#endif
```

>>> listaper.h

FdI UCM

Listas de objetos dinámicos

```
// Clase Lista - Implementación "listaper.cpp"
#include "listaper.h"
#include <iostream>
using namespace std;

Lista::Lista() : _cont(0) { }

Lista::~Lista() {
  for(int i = 0; i < _cont; i++)
    delete _array[i];
}

bool Lista::llena() const { return _cont == MAX; }

bool Lista::vacia() const { return _cont == 0; }

bool Lista::insertar(Persona* p) {
// true si se inserta; false si lista llena
  if(_cont == MAX) return false;
  _array[_cont] = p;  _cont++;  return true;
}
```

(*continúa*)

```
bool Lista::recuperar(int pos, Persona*& p) const {
// true con éxito; false si posición no válida
  if(pos < 1 || pos > _cont) return false;
  p = _array[pos-1];
  return true;
}


void Lista::mostrar() const {
  cout << "====================================" << endl;
  cout << "Elementos de la lista:" << endl << endl;
  for(int i = 0; i < _cont; i++) {
    _array[i]->mostrar();
    cout << endl;
  }
  cout << "====================================" << endl;
}
```



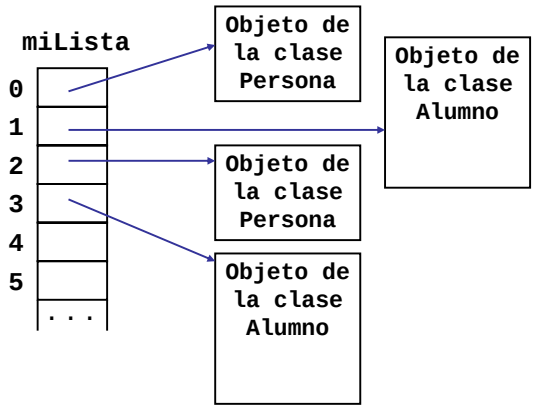
```
#include <iostream>
using namespace std;
#include "persona0.h"
#include "alumno0.h"
#include "listaper.h"

int main()
{
  Persona profe("666999F", 30, "Luis", "Hernández Yáñez");
  Persona* p;
  Alumno* al;
  Lista miLista;
  p = new Persona("223344K", 35, "Javier", "Lois Blaze");
  miLista.insertar(p);
  al = new Alumno(&profe, "123456J", 23, "Rosa", "Bose Iza", 2);
  miLista.insertar(al);
  miLista.mostrar();

  return 0;
}
```

persona0 y **alumno0** (**.h** y **.cpp**) son las versiones de las clases **Persona** y **Alumno** de la última unidad (atributo **_profesor** como relación entre **Alumno** y **Persona**).

Al ser ahora la lista un array de punteros a objetos, ya no se pierde información cuando se inserta un alumno, ya que se guarda un puntero al objeto, y todos los punteros ocupan lo mismo:



Como **Alumno** es subclase de **Persona**, la regla de compatibilidad de objetos permite asignar a un puntero a **Persona** un puntero a **Alumno**.

```
bool insertar(Persona* p) {
  ...
  _array[_cont] = p; _cont++; return true; }
  // _array es un array de punteros a Persona


miLista.insertar(al); // al es puntero a Alumno
```

Utilizando punteros hemos solucionado los problemas de tamaño.

Sin embargo, aunque ya no se pierde información y lo que se mantiene en la lista son personas y alumnos indistintamente (a través de punteros), seguimos teniendo problemas de vinculación.

Listas de objetos dinámicos

Los problemas de vinculación se evidencian en el método **mostrar()**, que hace lo siguiente:

```
...
for(int i = 0; i < _cont; i++)
  { _array[i]->mostrar(); cout << endl; }
```

Aunque algunas posiciones del array apunten a alumnos, al ser un array de punteros a personas se vincula por defecto cada mensaje **_array[i]->mostrar()** con el método **mostrar()** de la clase **Persona**.

El problema de vinculación todavía sigue estando presente.

```
====================================
Elementos de la lista:

223344K
Javier Lois Blaze
Edad: 35

123456J
Rosa Bose Iza
Edad: 23

====================================
```

¿Curso?

Vinculación dinámica

Cuando a través de un mismo puntero se puede hacer referencia a distintas clases de objetos se hace necesario incorporar en el programa el mecanismo de vinculación dinámica.

El mecanismo de vinculación dinámica determina la clase del objeto referenciado que recibe un mensaje para vincular correctamente el mensaje con el método de la clase del objeto.

p->*mensaje*()

¿Cuál es la clase del objeto apuntado por **p**? **X**

¿Existe en la clase **X** un método denominado ***mensaje()***?

Si es así, ejecutarlo.

Si no, buscarlo en las superclases.

Vinculación estática frente a vinculación dinámica

Vinculación estática

```
Persona* p;
...
p->mostrar();
```

p es un puntero a objetos de la clase **Persona**.

Se vincula el mensaje con el método **mostrar()** de la clase **Persona** (propio o heredado).

La vinculación se puede realizar en tiempo de compilación: *Vinculación anticipada*.

Es el mecanismo de vinculación utilizado por defecto.

Vinculación estática frente a vinculación dinámica

Vinculación dinámica

```
Persona* p;
...
p->mostrar();
```

p es un puntero a objetos de la clase **Persona**.

p puede apuntar a objetos de la clase **Persona** o de cualquier subclase de **Persona**.

¿A qué clase de objeto está apuntando en ese momento?

- ✓ Si apunta a un objeto de la clase **Persona**, ejecutar el método **mostrar()** de la clase **Persona**.
- ✓ Si apunta a un objeto de la clase **Alumno**, ejecutar el método **mostrar()** de la clase **Alumno**.

Esta vinculación sólo se puede realizar en tiempo de ejecución (*vinculación postergada*).

Vinculación estática frente a vinculación dinámica

Vinculación dinámica (postergada, en tiempo de ejecución)

```
Persona unaPersona;
Alumno unAlumno;
Persona* p;
int op;
...
cout << "¿Qué objeto (1=Persona, 2=Alumno)? "; cin >> op;
if(op == 1) p = &unaPersona;
else p = &unAlumno;
p->mostrar();
```

Hasta que se ejecute el programa
no podemos saber qué opción elegirá el usuario.

⇒ no podemos saber si **p** apuntará al objeto de clase **Persona** o al objeto de clase **Alumno**.

⇒ no podemos saber con qué método se ha de vincular el mensaje **mostrar()** (se vinculará dinámicamente en tiempo de ejecución).

Vinculación dinámica y métodos virtuales

El mensaje **mostrar()** del ejemplo anterior puede *virtualmente* corresponderse con el método **mostrar()** de la clase **Persona** o con el método **mostrar()** de la clase **Alumno**.

Esa *virtualidad* debe hacerse explícita en el programa, indicando cuáles son los métodos que se ven afectados por la necesidad de vinculación dinámica, cuáles son los *métodos virtuales*.

Definiendo en las clases algunos métodos como virtuales se fuerza la incorporación del mecanismo de vinculación dinámica en los programas.

Sólo han de establecerse como virtuales los métodos que se puedan ver afectados por problemas de vinculación que se deben resolver en tiempo de ejecución.

En nuestro ejemplo de **Persona**s y **Alumno**s el único método que debe ser virtual (en ambas clases) es el método **mostrar()**. (En realidad hay otro método más que debe ser virtual.)

Métodos (funciones miembro) virtuales

Para establecer un método como virtual basta con colocar al principio de la declaración de la correspondiente función miembro la palabra reservada **virtual**:

```
class Persona {
  ...
  virtual void mostrar();
  ...
};

class Alumno : public Persona {
  ...
  void mostrar();
  ...
};
```

Los métodos virtuales se encuentran en las superclases.

Si la clase **Alumno** no tiene subclases no es necesario que el método **mostrar()** sea virtual.

virtual sólo se pone en el prototipo

Haciendo que el método **mostrar()** sea virtual los problemas de vinculación desaparecen y el programa de lista de **Persona**s y **Alumno**s funciona ya correctamente.

Listas de objetos polimórficas: solución final

Ya tenemos todo lo necesario para construir listas polimórficas sin problemas de tamaño ni de vinculación: los punteros a objetos y los métodos virtuales.

El programa **pruebaper2.cpp** usa otra versión de **Lista** (**listaper2.h**) que a su vez utiliza una versión de la clase **Persona** con métodos virtuales.

```
====================================
Elementos de la lista:

223344K
Javier Lois Blaze
Edad: 35

123456J
Rosa Bose Iza
Edad: 23
Curso: 2
Profesor:
666999F
Luis Hernández Yáñez
Edad: 30

====================================
```

>>> pruebaper2.cpp

Polimorfismo

La regla de compatibilidad de objetos, los punteros a objetos y los métodos virtuales nos permiten incorporar en los programas orientados a objetos el *polimorfismo*.

Cuando no podemos estar seguros de la clase de objeto a la que apunta un puntero cuando se pasa un mensaje, es porque el puntero puede hacer referencia (apuntar) a objetos de varias clases.

```
Persona* p;
...
p->mostrar();
```

El objeto que referencia **p** puede ser una **Persona** o un **Alumno**.

El objeto que referencia **p** unas veces será una **Persona** y otras veces será un **Alumno**.

El objeto que referencia **p** puede tomar distintas formas (**Persona** /**Alumno**).

El objeto que referencia **p** es *polimórfico*.

Destructores virtuales

Cuando entra el juego el polimorfismo los destructores de las superclases deben ser virtuales, para asegurar que se destruyan correctamente los objetos de las subclases.

Veamos qué ocurre SIN destructores virtuales:

```
class Super {
public:
  Super {cout << "Constructor de Super" << endl;}
  ~Super {cout << "Destructor de Super" << endl;}
};

class Sub : public Super {
public:
  Sub {cout << "Constructor de Sub" << endl;}
  ~Sub {cout << "Destructor de Sub" << endl;}
};

int main() {
  Super* p;
  p = new Sub();
  delete p; ...
```

```
Constructor de Super
Constructor de Sub
Destructor de Super
```

No se ejecuta el destructor de la subclase

Destructores virtuales

Si hacemos que el destructor de la superclase sea virtual, el objeto apuntado por **p** (de clase **Sub**) se destruye correctamente:

```
class Super {
public:
  Super {cout << "Constructor de Super" << endl;}
  virtual ~Super
    { cout << "Destructor de Super" << endl; }
};
class Sub : public Super {
...
}
int main()
{
    Super* p;
    p = new Sub();
    delete p;
    return 0;
}
```

```
Constructor de Super
Constructor de Sub
Destructor de Sub
Destructor de Super
```


El problema de la duplicación de listas polimórficas

Para incorporar el constructor de copia y el operador de asignación en la clase **Lista** anterior debemos resolver un problema que surge cuando se plantea la duplicación de listas polimórficas.

La lista es básicamente un array declarado como de punteros a objetos de una superclase (**Persona**). La regla de compatibilidad de objetos nos permite apuntar a objetos de esa clase o a objetos de cualquiera de sus subclases (**Alumno**).

Entonces, cuando nos planteamos crear una copia de ese array nos damos cuenta de que lo que tenemos que hacer es simplemente colocar en cada posición del nuevo array un puntero a una copia del objeto apuntado por la correspondiente posición del array de partida (tenemos que hacer copias para que las listas no compartan memoria).

Muy bien. Suponiendo que hay más de un elemento en la lista, empezamos por la primera posición (**_array[0]**). ¿A qué apunta?

El problema de la duplicación de listas polimórficas

_array[0] puede apuntar a una **Persona** o a un **Alumno**.
Eso es todo lo que sabemos. No tenemos forma de distinguir si se trata de una clase u otra de objeto.

¿Cómo podemos saber cuál es la clase de un objeto?

Los compiladores proporcionan medios para poder conocer cuál es la clase del objeto apuntado por un puntero durante la ejecución de un programa. Sin embargo, son herramientas que no son en absoluto necesarias, ya que si tenemos que llegar a recurrir a ellas, entonces nuestro diseño orientado a objetos no es bueno. Si el diseño es bueno, no necesitaremos nunca preguntar por la clase de un objeto.

Entonces, ¿cómo podemos crear una copia del objeto apuntado por **_array[0]** si no sabemos cuál es su clase concreta y por tanto no sabemos qué poner a continuación de **new**?

Sencillamente pidiéndoselo al propio objeto. Él sabe perfectamente cuál es su clase y, por tanto, qué mensajes entiende.

El problema de la duplicación de listas polimórficas

En la POO se debe delegar todo aquello que tenga que ver con cada clase de objetos en la propia clase de objetos. Incluso las copias.
Si queremos una copia de un objeto de una clase, lo que necesitamos es que la clase esté preparada para darnos esa copia.

Es decir, en aquellas clases de las que vayamos a necesitar crear copias de objetos dinámicos suyos a través de punteros polimórficos, incluiremos un método que se encargue de proporcionar una copia.

A estos métodos los llamaremos siempre **clon()**, de forma que den a entender lo que hacen. Lo que devolverá el método será un puntero a una copia dinámica del objeto receptor (un *clon* suyo).

Pero como sólo los punteros a la superclase pueden apuntar a objetos de cualquiera de las clases de la jerarquía, la declaración del método **clon()** en todas esas clases será:

```
Superclase* clon() const; // No modifica al receptor
```


El problema de la duplicación de listas polimórficas

Por ejemplo, tanto en la clase **Persona** como en su subclase **Alumno** la declaración será:

```
Persona* clon() const;
```

Así, bastará siempre un puntero a **Persona** para recoger la copia.

Por otro lado, como el método *de clonación* debe llamarse igual en todas las clases de la jerarquía, se verá afectado por la vinculación dinámica, por lo que debemos declararlo como virtual:

```
virtual Persona* clon() const;
```

La implementación de estos métodos clonadores en las clases **Persona** y **Alumno** se muestra en las páginas siguientes.

La clase **Persona** con método clonador

```
...
class Persona {
public:
  ...
  virtual ~Persona(); // Destructor
  // Método clonador
  virtual Persona* clon() const;
  // Mutadores:
  ...
  // Visualizador:
  virtual void mostrar() const;
  ...
```

```
Persona* Persona::clon() const {
  Persona* p = new Persona();
  *p = *this;
  return p;
}
```

La clase **Alumno** con método clonador

```
...
class Alumno : public Persona {
public:
  ...
  ~Alumno(); // Destructor
  // Método clonador
  Persona* clon() const;
  ...
```

```
Persona* Alumno::clon() const {
  Alumno* p = new Alumno();
  *p = *this;
  return p;
}
```


Duplicación de la lista polimórfica

```
Lista::Lista(const Lista& otra) {
  _cont = otra._cont;
  for(int i = 0; i < _cont; i++)
    _array[i] = otra._array[i]->clon();
}

Lista& Lista::operator=(const Lista& otra) {
  // Primero eliminamos los elementos de la lista
  for(int i = 0; i < _cont; i++)
    delete _array[i];
  _cont = otra._cont;
  for(int i = 0; i < _cont; i++)
    _array[i] = otra._array[i]->clon();
  return *this;
}
```

!

Clases abstractas

En ocasiones crearemos clases de las que no se van a crear ejemplares nunca y cuyo cometido es establecer pautas para las interfaces de las subclases que se deriven de ella.

Tales clases se identifican como *clases abstractas*.

Una clase abstracta define:

- ✓ Atributos y métodos comunes a todas las subclases.
- ✓ Métodos que deben estar presentes en todas las subclases, pero cuya implementación corresponde a cada subclase.

Siendo abstracta, no tendrá sentido crear ejemplares de la clase y no tendrá sentido implementar algunos métodos (no se sabe cómo).

Si se quiere obligar a las subclases a que implementen determinados métodos y no tiene sentido implementar esos métodos en la clase, se utilizan métodos virtuales puros.

```
virtual void mostrar() = 0;
```


Clases abstractas

Una clase abstracta debe tener al menos un método virtual puro.

- ✓ C++ no permite crear ejemplares de una clase abstracta.
- ✓ C++ obliga a que las subclases tengan implementados los métodos de la superclase abstracta que estén definidos como virtuales puros.
- ✓ Si una subclase de la clase abstracta no implementa uno de sus métodos virtuales puros, entonces se considera también abstracta.

```
class Forma {
public:
  virtual void rotar(int) = 0;
  virtual void dibujar() = 0;
  virtual bool es_cerrada() = 0;
};
```

Clase abstracta

(*continúa*)

Clases abstractas

```
class Punto { ... };

class Circulo : public Forma { // No abstracta
public:
  void rotar(int) { } // Operación sin efecto
  void dibujar();
  bool es_cerrada() { return true; }
private:
  Punto _centro;
  int _radio;
};

void Circulo::dibujar() { ... }

class Linea : public Forma { // No abstracta
public:
  void rotar(int);
  void dibujar();
  bool es_cerrada() { return false; }
```

Obviamos la mayoría de los métodos de las clases para simplificar el ejemplo y centrarlo en lo más significativo

(*continúa*)

Clases abstractas

```
private:
   Punto _inicio, _fin;
};

void Linea::rotar(int grados) { ... }

void Linea::dibujar() { ... }

class Poligono : public Forma { // ABSTRACTA
public: // No redefine los métodos rotar() y dibujar()
   bool es_cerrada() { return true; }
};

class Poligono_irregular : public Poligono {
public: // No abstracta
   void rotar(int);
   void dibujar();
private:
   Punto _puntos[10];
};
```

Las clases abstractas se deben utilizar para establecer requisitos de interfaz de un conjunto de subclases

Etcétera ...