Clase 6

Sobre la Programación Orientada a Objetos II

Prof. Andrés Emilio Arcia Moret

¿Qué es la OxO?

- En principio, la OxO es una técnica de programación, un paradigma para escribir buenos programas para cierta clase de problemas.

¿Cómo programar en C++?

1) Obtener una buena comprensión del problema.
2) Identificar los conceptos claves que están dentro de la solución.
3) Expresar la solución en un lenguaje - C++ -

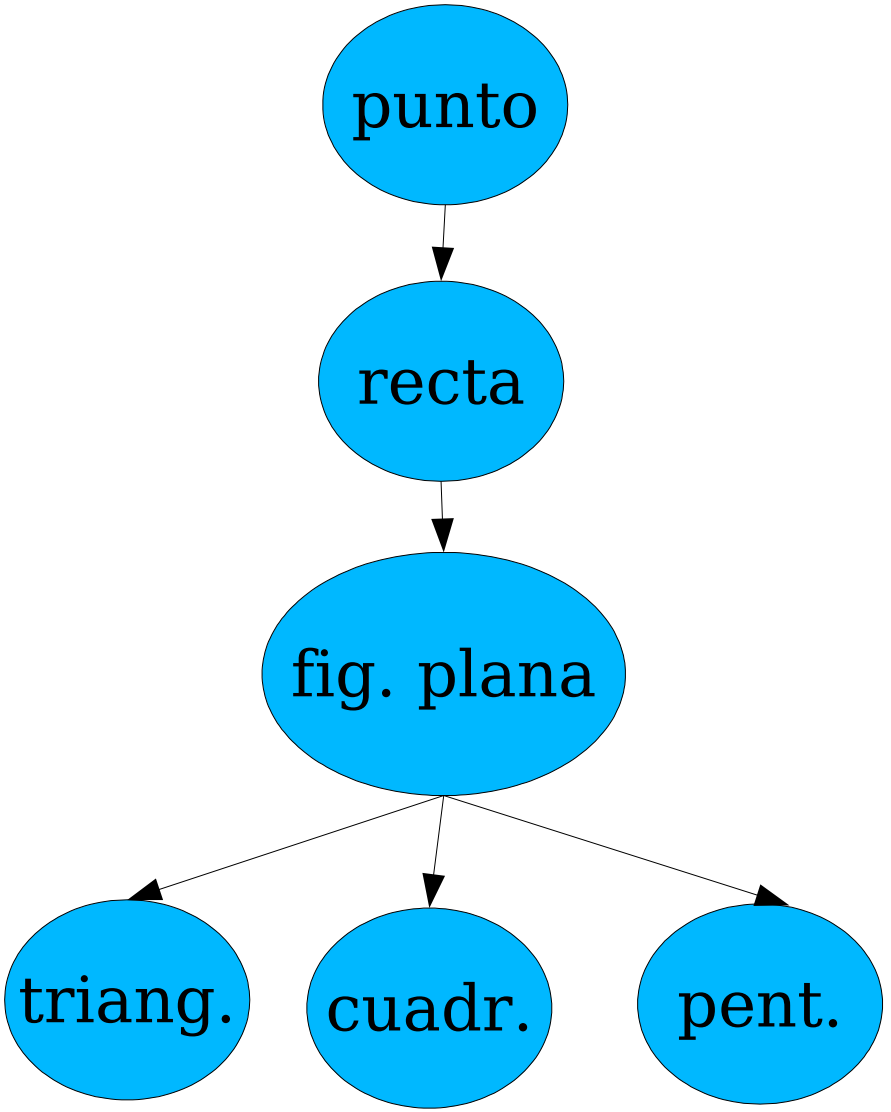
La clave para escribir buenos programas es escribir clases tal que c/u represente un solo concepto.

Siempre será bueno tardarse en 1 y 2, luego 3 será más llevadero.

Aspectos generales de Programación

- Una de las herramientas más poderosas para diseñar sistemas es el orden jerárquico.
- Un orden jerárquico corresponde a la organización de conceptos relacionados en la forma de un organigrama (forma de árbol). El concepto más general está en la raíz.

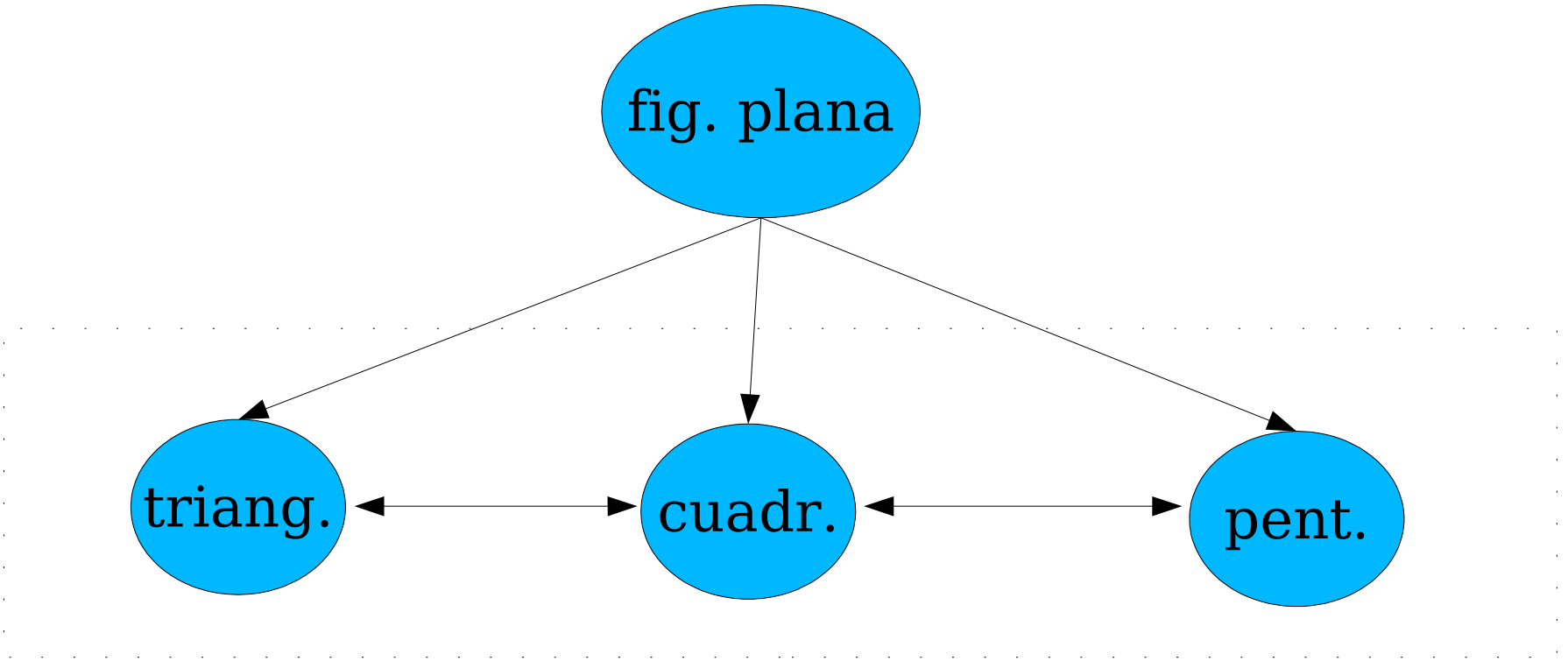
Ejemplo



Aspectos generales de la programación

- Aunque no todo el tiempo se pueden organizar las ideas de esta forma. Puede ser que las ideas sean mutuamente dependientes.
- Cuando hay ideas mutuamente dependientes, hay una relación horizontal.
- Para decifrar lo que será un grafo de dependencia, lo mejor es la separación entre la interfaz y la implantación.

Ejemplo de Dependencia Horizontal



composición de figuras

¿Cómo se escribe un buen Código?

- Conozca plenamente lo que quiere expresar. Esto implica conocer bien el problema y la solución que se propone.
- ¡Practique! Para ser un buen programador, hay que empezar por imitar a los buenos programadores...

Concejos para programar OxO

- Casi nunca se obtiene un buen programa en el primer intento. Hay que tener paciencia e intentar varias veces.
- Para ser crítico hay que saber probar el programa y estar dispuesto a cambiarlo.

Concejos para programar OxO

- Programar es representar las ideas de una solución a un problema. Lo importante entonces es la estructura que tendrán esas ideas. El mayor problema en la programación: la escala.
  - Si puede pensar algo como una **idea** separada, hagalo una clase.
  - Si puede pensar algo como una **entidad** separada, hagalo un objeto de una clase.
  - Si dos clases tienen una interfaz común, hagala interfaz de una clase abstracta.
  - Si en la implantación de un par de clases hay código común, hagalo como clase base.

Concejos para programar OxO

  - Si una clase es un contenedor de objetos, entonces hagalo plantilla.
- Si una función implanta un algoritmo para un **contendor**, hagala una función **plantilla**. Así, funcionara para una familia de contenedores.
- Si un conjunto de clases, plantillas, etc., están lógicamente relacionadas, pongalas en un espacio de nombres.

Concejos para programar OxO

- En ciertas circunstancias (fuera del ambito matemático, por ejemplo):
  - No utilice data global.
  - No use funciones globales.
  - No use datos miembros públicos.
  - No use la amistad, excepto si se quiere evitar el uso de data global o los datos miembros públicos.
  - No use tipo enumerado en una clase, use funciones virtuales.
  - No use funciones inline, excepto cuando desee hacer una optimización significativa.

Clases

- Una clase es un conjunto de datos y operaciones sobre esos datos, pero con añadiduras para hacer la abstracción mucho más poderosa.

```
class primer_ejemplo {

int dato1;

float dato2;

public:

int ver_dato1()  { return dato1; } };
```

Estos datos no son “abstractos” son tan reales como los `int` y los `float`. Doug McIlory.

Summary--what's most important.

To put my strongest concerns into a nutshell:

1. We should have some ways of connecting programs like garden hose--screw in another segment when it becomes when it becomes necessary to massage data in another way. This is the way of IO also.

2. Our loader should be able to do link-loading and controlled establishment.

3. Our library filing scheme should allow for rather general indexing, responsibility, generations, data path switching.

4. It should be possible to get private system components (all routines are system components) for buggering around with.

M. D. McIlroy

October 11, 1964

What did Doug mean, and what came from the thoughts?

Point 1 is the interesting one. It provides the historical background for Doug's encouragement of the Unix pipe notation. The linked paper gives appropriate credit; in interviews, Doug has been explicit in saying that he very nearly exercised managerial control to get pipes installed.

Point 2 is a bit mysterious without further context. It is possible to read it as advocating mechanisms (dynamic linking) that were pioneered in Multics.

Point 3 is almost certainly the one that still bugs Doug. All sorts of mechanisms and utilities are around and used (source code control, registries, WWW search engines, and on and on), but the problem of indexing and finding relevant information is tougher today than ever before, even on one's own hard disk, let alone the WWW.

One might argue that Point 4 is to some extent under control, depending on what Doug was really getting at. One thinks about the dynamic linking mechanism combined with search paths in Multics (close to the time of the note) and its offspring in more recent systems. Possibly one could create a connection to the free software movement, but this would be tenuous at best; I suspect that he was thinking "how can I test this seemingly monolithic program against the sin routine I've just written and want to test?".

Point 1's garden hose connection analogy, though, is the one that ultimately whacked us on the head to best effect.

Clases

- Observe la manera en que se crea el nuevo tipo de dato: primer_ejemplo.
- Posee un par de datos: uno entero y otro real que conforman los datos y una sola operación pública: ver_dato1.
- Así como existe la característica pública, existe la privada que no deja exportar el contenido.

Clases

- Una clase con características privadas, permite encapsular o esconder al usuario datos que no son de su incumbencia.

```
class persona {

char nombre[30];

char apellido[30];

TIPOS tipo_persona;

int sueldo;

public:

void set_nombre(char * nom);

... };
```

Herencia

class estudiante : public persona {

... };

Un conjunto de datos, por omisión es privado. En el ejemplo anterior, la clase estudiante **hereda**, publicamente de la clase persona. Puede leerse: se deriva de, implementa, es un subtipo de...

Todos los métodos y datos de la clase persona, son accessibles desde las instancias de la clase estudiante.

class estudiante : private persona {

... };

Los métodos de la clase persona ahora no son accesibles desde las instancias de la clase estudiante.

Por ejemplo el tipo de dato tipo_persona, no deberia ser accesible desde persona, pues se fija desde el constructor de la clase estudiante.

Herencia

- Cuando hay herencia pública, los permisos de la clase original se preservan.

```
class A {

private:

int a, b;

public:

void set_a(int _a);

void set_b(int _b);

};

class B : public class A { //contenido de B };
```

Herencia

- Los datos heredados de la clase A siguen teniendo los mismo permisos en B.
- Si la herencia es privada, entonces cambia. Los datos publicos en la clase base serán privados en la clase heredada y los datos privados pasan a ser protegidos (inaccesibles) en la clase heredada.

Herencia

class B : private class A {

// datos + métodos de B

B() // constructor por omisión queda reemplazado

{ set_a(5); // CORRECTO desde dentro de la clase }

};

Luego:

B objB;

No podremos hacer:

objB.set_a(5); // ERROR, pues set_a es privada.

Clases

- Una clase esta básicamente compuesta de:
  - Constructor: inicializa la data de la clase y aparta los recursos necesarios. También hay variantes en la construcción para comodidad del cliente.
  - Un constructor tiene el mismo nombre de la clase.

Constructor

```
class Fecha {
int m, d, a;
public:
   Fecha(int, int, int);    // se introduce dia, mes, año
   Fecha(int, int); // día, mes y el año presente
   Fecha(float, float); // dia y mes partiendo de un par de
    flotante.
   Fecha(int); // día, y el mes y año presente
   Fecha(); // la fecha de hoy, extraida del sistema
   Fecha(int, char *); // dia en número, mes en letras
};
```

Clases Públicas

- Recuerde que una estructura, puede contener funciones que operan sobre la data.
- Entonces, un “struct” es una clase con todos sus miembros públicos.

```
struct ClaseUno {
int a; float b;
void set_a(int _a); };
class ClaseUno { // Esta clase es completamente equivalente a la anterior
public:
int a; float b;
void set_a(int _a); };
```

Destructores

- Cuando en una clase se han apartado datos de forma dinamica: malloc o new, deben liberarse al destruir el objeto: free o delete.

```
class ejemploMemoria {

int * ArrayInt;

public:

ejemploMemoria(int tamano)

{  ArrayInt = new int[tamano]; }

~ejemploMemoria()      // Observe la sintaxis del destructor

{  delete [] ArrayInt; }  // Observe el operador
```

Sobre Constructores y Destructores

- Un objeto se construye automáticamente cuando se declara una instancia y se destruye cuando finaliza el ambito.
- Un objeto se crea de forma dinámica con new y se libera con delete. El operador new invoca al constructor.
- Un objeto puede crearse dentro de otros objetos, de forma heredada y como miembro.
- Un objeto puede crearse de forma global y se crea con el inicio de la aplicación.

Operadores de Asignación de Memoria

- new aparta memoria para objetos. Con new se invoca al constructor del objeto o conjunto de objetos apartados.
- delete libera la memoria apartada con new.
- Para liberar memoria de arreglos se utiliza delete [].

Arreglo de Objetos

- Los objetos pueden estar en arreglos tal y como el resto de los tipos de datos:

```
class Cubeta {

int j, k;

public:

Cubeta(int _j, int _k) : j(_j), k(_k) { }

};

Cubeta * arrayCubeta = new Cubeta[10];

Cubeta * arrayCubeta2 = new Cubeta[10](0,0);
```

Concejos

- Represente los conceptos como clases (recuerde las ideas)
- Utilice datos públicos solo cuando su variabilidad es irrelevante.
- Un dato concreto (int, float, bool) es la clase más simple. Utilicelos mientras sea posible.
- Haga métodos de clases solo si se necesita el acceso a los datos
- Los constructores establecen invariantes en las clases
- Si un constructor adquiere un recurso, el destructor debe liberarlo.
- Si una clase tiene un puntero, este necesita un constructor copia y asignación definidos.
- Hay que tener cuidado con los constructores por omisión.