Tema 7: Introducción a la programación orientada a objetos

Programación 2

Curso 2013-2014

Notas

Índice

Notas

Definición

- La POO es un paradigma de programación que usa objetos y sus interacciones para diseñar aplicaciones y programas informáticos.
- La aplicación entera se reduce a un conjunto de objetos y sus relaciones.
- C++ es un lenguaje orientado a objetos, aunque también permite programación imperativa (procedural).
- Prepárate para cambiar la mentalidad y el enfoque de la programación tal como lo hemos visto hasta ahora.

Notas

Clases y objetos

- En Programación 2 ya hemos usado clases y objetos.

```
int i;  // Declaramos una variable i de tipo int
```

```
string s; // Declaramos un objeto s de clase string
```

- Las clases o tipos compuestos son similares a los tipos simples aunque permiten muchas más funcionalidades.
- Una clase es un modelo para crear objetos de ese tipo.
- Un objeto de una determinada clase se denomina una instancia de la clase (`s` es una instancia de `string`).

Notas

Diferencias entre tipos simples de datos y clases

- Un registro (tipo simple) se puede considerar como una clase ligera que sólo almacena datos visibles desde fuera.
- Básicamente, una clase es similar a un registro, pero añadiendo funciones (clase = datos + métodos).
- También permite controlar qué datos son visibles (parte pública) y cuáles están ocultos (parte privada).
- Una clase contiene datos y una serie de funciones que manipulan esos datos llamadas funciones miembro.

Notas

Sintaxis (1/2)

- Ejemplo de registro:

```
struct Fecha {
   int dia;
   int mes;
   int anyo;
};
```

- Equivalente sencillo/cutre de clase:

```
class Fecha {
  public:
     int dia;
     int mes;
     int anyo;
}; // Ojo: el punto y coma del final es necesario
```

- Si no se indica lo contrario (`public`), todos los miembros de la clase son privados.

Notas

Sintaxis (2/2)

- Acceso directo a elementos como en un registro:

```
Fecha f;
f.dia=12;
```

- Pero en una clase normalmente **no** conviene acceder directamente a los elementos. Para modificar los datos, se usan métodos.
- En el ejemplo anterior, `f.dia=100` no daría error. Pero con métodos podemos controlarlo.

```
class Fecha {
  private: // Solo accesible desde metodos de la clase
     int dia;
     int mes;
     int anyo;
  public:
     bool setFecha(int d, int m, int a) { ... };
};
```

Notas

Conceptos básicos

Principios en los que se basa el diseño orientado a objetos:

- Abstracción
- Encapsulación
- Modularidad
- Herencia
- Polimorfismo

Notas

Abstracción

- La abstracción denota las características esenciales de un objeto y su comportamiento.
- Cada objeto puede realizar tareas, informar y cambiar su estado, y comunicarse con otros objetos en el sistema sin revelar cómo se implementan estas características.
- El proceso de abstracción permite seleccionar las características relevantes dentro de un conjunto e identificar comportamientos comunes para definir nuevas clases.
- El proceso de abstracción tiene lugar en la fase de diseño.

Notas

Encapsulación

- La encapsulación significa reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción.
- La interfaz es la parte del objeto que es visible para el resto de los objetos (la parte pública). Es el conjunto de métodos (y a veces datos) del cual disponemos para comunicarnos con un objeto.
- Cada objeto oculta su implementación y expone una interfaz.
- Interfaz: **Qué** hace un objeto. Implementación: **Cómo** lo hace.
- La encapsulación protege a las propiedades de un objeto contra su modificación, solamente los propios métodos internos del objeto pueden acceder a su estado.

Notas

Modularidad

- Se denomina *modularidad* a la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos) tan independientes como sea posible.
- Estos módulos se pueden compilar por separado, pero tienen conexiones con otros módulos.
- Generalmente, cada clase se implementa en un módulo independiente, aunque clases con funcionalidades similares también pueden compartir módulo.

Notas

Herencia (no en Programación 2)

- Las clases se pueden relacionar entre sí formando una jerarquía de clasificación. Por ejemplo, un coche (`subclase`) es un vehículo (`superclase`).
- Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen.
- La herencia facilita la organización de la información en diferentes niveles de abstracción.
- Los objetos derivados pueden compartir (y extender) su comportamiento sin tener que volver a implementarlo.
- Cuando un objeto hereda de más de una clase se dice que hay herencia múltiple.

Notas

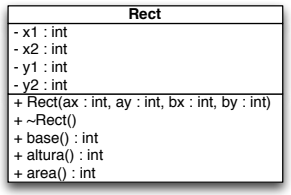
Polimorfismo (no en Programación 2)

- El polimorfismo es la propiedad según la cual una misma expresión hace referencia a distintas acciones. Por ejemplo, el método `desplazar()` puede referirse a acciones distintas si se trata de un avión o de un coche.
- Comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre.
- Las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos.

```
Animal *a = new Perro;
Animal *b = new Gato;
Animal *c = new Gaviota;
```

Notas

Ejemplo de clase (1/2)




```
// Rect.h (declaracion de la clase)
class Rect
{
   private:
        int x1, y1, x2, y2;
   public:
        Rect(int ax, int ay, int bx,int by); // Constructor
        ~Rect();  // Destructor
        int base();
        int altura();
        int area();
};
```

Notas

Ejemplo de clase (2/2)

```
// Rect.cc (implementacion de metodos)
Rect::Rect(int ax, int ay, int bx, int by) {
   x1=ax;
   y1=ay;
   x2=bx;
   y2=by;
}

Rect::~Rect() { }

int Rect::base() { return (x2-x1); }
int Rect::altura() { return (y2-y1); }
int Rect::area() { return base()*altura(); }
```

```
// main.cc
int main()
{
     Rect r(10,20,40,50);
     cout << r.area() << endl;
}
```

Notas

Declaraciones inline (1/2)

- Los métodos con poco código también se pueden implementar directamente en la declaración de la clase.

```
// Rect.h (declaracion de la clase)
class Rect
{
   private:
        int x1, y1, x2, y2;
   public:
        Rect(int ax, int ay, int bx,int by);
        ~Rect() {};  // Inline
        int base() { return (x2-x1); };  // Inline
        int altura() { return (y2-y1); }; // Inline
        int area();
};
```

- Es más eficiente ya que, cuando se compila, el código generado para las funciones `inline` se inserta en el punto donde se invoca a la función, en lugar de hacerlo en otro lugar y hacer una llamada.

Notas

Declaraciones inline (2/2)

- Las funciones inline también se pueden implementar fuera de la declaración de clase (en el `.cc`)

```
inline int Rect::base()
{
  return (x2-x1);
}
```

Notas

Accesores

- Por el principio de encapsulación, no es conveniente acceder directamente a los datos miembro de una clase.
- Lo normal es definirlos como private y, para acceder a ellos, implementar funciones set/get/is (llamadas `accesores`).

- Los accesores `set` nos permiten controlar que los valores de los atributos sean correctos.

Notas

Forma canónica

- Todas las clases deben implementar al menos cuatro métodos importantes:
  - Constructor
  - Operador de asignación (No en Programación 2)
  - Constructor de copia
  - Destructor
- Son las operaciones definidas de oficio en C++

Notas

Constructores (1/3)

- Las clases suelen tener al menos un método constructor y otro destructor.
- El constructor se invoca automáticamente cuando se crea un objeto de la clase, y el destructor cuando se termina de usar.
- Si no definimos un constructor, el compilador creará uno por defecto sin parámetros y que no hará nada. Los datos miembros de los objetos declarados así contendrán basura.
- En una clase puede haber varios constructores con parámetros distintos (el constructor puede *sobrecargarse*). La sobrecarga es un tipo de polimorfismo.

Notas

Constructores (2/3)

- Ejemplos de constructor:

```
Fecha::Fecha() {
  dia=1;
  mes=1;
  anyo=1900;
}
Fecha::Fecha(int d, int m, int a) {
  dia=d;
  mes=m;
  anyo=a;
}
```

- Llamadas al constructor:

```
Fecha f;
Fecha f(10,2,2010);
```

```
Fecha f(); // INCORRECTO (error frecuente)
```

Notas

Constructores (3/3)

- Es preferible usar la inicialización en lugar de asignación al ser menos costoso y más predecible inicializar objetos en el momento de su creación.

```
Fecha::Fecha(int d,int m,int a) : dia(d) mes(m),anyo(a)
{ }
```

- Constructores con parámetros por defecto (solo se ponen en .h):

```
Fecha(int d=1,int m=1,int a=1900);
```

Con este constructor podemos crear un objeto de varias formas:

```
Fecha f;
Fecha f(10,2,2010);
Fecha f(10);  // dia=10
```

Notas

Operador de asignación

- Podemos hacer una asignación directa de dos objetos (sin usar constructores de copia).

```
Fecha f1(10,2,2011);
Fecha f2;
f2=f1;  // Copia directa de valores de los datos miembro
```

- Podemos redefinir el operador = para nuestras clases si lo consideramos necesario (No en Programación 2).

Notas

Constructores de copia (1/2)

- De modo similar a la asignación, un constructor de copia crea un objeto a partir de otro objeto existente.

```
// Declaracion
Fecha(const Fecha &f);

// Implementacion
Fecha::Fecha(const Fecha &f) :
   dia(f.dia), mes(f.mes), anyo(f.anyo) {}
```

Notas

Constructores de copia (2/2)

- El constructor de copia se invoca automáticamente cuando...
  - Una función devuelve un objeto
  - Se inicializa un objeto cuando se declara

```
Fecha f2(f1);
Fecha f2 = f1;
f1=f2; // Aqui NO se invoca al constructor, sino a =
```

  - Un objeto se pasa por valor a una función

```
void funcion(Fecha f1);
funcion(f1);
```

- Si no se especifica ningún constructor de copia, el compilador crea uno por defecto con el mismo comportamiento que el operador =

Notas

Destructores (1/2)

- Todas las clases necesitan un destructor (si no se especifica, el compilador crea uno por defecto).
- Un destructor debe liberar los recursos (normalmente, memoria dinámica) que el objeto esté usando.
- Es una función miembro con igual nombre que la clase y precedido por el carácter ~
- Una clase sólo tiene una función destructor que no tiene argumentos y no devuelve ningún tipo.
- El compilador llama automáticamente a un destructor del objeto cuando acaba su ámbito. También se invoca al destructor al hacer `delete`. Se puede invocar explícitamente: `f.~Fecha();`
- Aunque se puede, **nunca** se debe invocar explícitamente.

Notas

Destructores (2/2)

- Ejemplo:

```
// Declaracion
~Fecha();

// Implementacion
Fecha::~Fecha() {
   // Liberar la memoria reservada (nada en este caso)
}
```

- Importante: El destructor de un objeto invoca implícitamente a los destructores de todos sus atributos.

Notas

Atributos y métodos de clase (1/2)

- Los atributos y métodos de clase también se llaman `estáticos`. Se representan subrayados en los diagramas UML.
- Los atributos de clase tienen el mismo valor para todos los objetos de la clase. Son como variables globales para la clase.
- Los métodos de clase sólo pueden acceder a atributos de clase.

```
class Fecha {
  public:
    static const int semanasPorAnyo = 52;
    static const int diasPorSemana = 7;
    static const int diasPorAnyo = 365;
    static string getFormato();
    static boolean setFormato(string);
  private:
    static string cadenaFormato;
};
```

Notas

Atributos y métodos de clase (2/2)

- Cuando el atributo estático no es un tipo simple o no es constante, debe declararse en la clase pero tomar su valor fuera de ella.

```
// Fecha.h (dentro de la declaracion de la clase)
static const string findelmundo;

// Fecha.cc
const string Fecha::findelmundo="2012";
```

Notas

El puntero `this`

- El puntero `this` es una pseudovariable que no se declara ni se puede modificar.
- Es un argumento implícito que reciben todas las funciones miembro (excluyendo funciones `static`).
- Apunta al objeto receptor del mensaje. Suele omitirse para acceder a los atributos mediante funciones miembro.
- Es necesario cuando queremos desambiguar el nombre del parámetro o cuando queremos pasar como argumento el objeto a una función anidada.

```
void Fecha::setDia (int dia) {
    // dia=dia; ERROR: ambiguo
    this->dia=dia;
    cout << this->dia << endl;
}
```

Notas

Métodos constantes

- Los métodos que no modifican los atributos del objeto se denominan métodos constantes.

```
int Fecha::getDia() const { // Metodo constante
    return dia;
}
```

- En un objeto constante no se puede invocar a métodos no constantes. Por ejemplo, este código no compilaría:

```
int Fecha::getDia() {
   return dia;
}

int main() {
   const Fecha f(10,10,2011);
   cout << f.getDia() << endl;
}
```

- Los métodos `get` deben ser constantes.

Notas

Funciones amigas (`friend`)

- La parte privada de una clase sólo es accesible desde:
  - Métodos de la clase
  - **Funciones amigas**
- Una función amiga no pertenece a la clase pero puede acceder a su parte privada.

```
class MiClase {
  friend void unaFuncionAmiga(int, MiClase&);
public:
  //...
private:
   int datoPrivado;
};
```

```
void unaFuncionAmiga(int x, MiClase& c) {
  c.datoPrivado = x; // OK
}
```

Notas

Sobrecarga de los operadores de entrada/salida (1/3)

- Podemos sobrecargar las operaciones de entrada/salida de cualquier clase:

```
MiClase obj;
cin >> obj;  cout << obj;
```

- El problema es que no pueden ser funciones miembro de MiClase porque el primer operando no es un objeto de esa clase (es un `stream`).
- Los operadores se sobrecargan usando funciones amigas:

```
friend ostream& operator<< (ostream &o, const MiClase& obj);
friend istream& operator>> (istream &o, MiClase& obj);
```

Notas

Sobrecarga de los operadores de entrada/salida (2/3)

- Declaración

```
class Fecha {
 friend ostream& operator<< (ostream &os, const Fecha& obj);
 friend istream& operator>> (istream &is, Fecha& obj);

 public:
   Fecha (int dia=1, int mes=1, int anyo=1900);
   ...
 private:
   int dia, mes, anyo;
};
```

Notas

Sobrecarga de los operadores de entrada/salida (3/3)

- Implementación

```
ostream& operator<< (ostream &os, const Fecha& obj) {
  os << obj.dia << "/" << obj.mes << "/" << obj.anyo;
  return os;
}
```

```
istream& operator>> (istream &is, Fecha& obj) {
  char dummy;
  is >> obj.dia >> dummy >> obj.mes >> dummy >> obj.anyo;
  return is;
}
```

Notas

Relaciones entre objetos

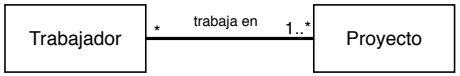
- Principales tipos de relaciones entre objetos y clases

Entre objetos	Asociación	
	Agregación	
	Composición	
	Uso	
Entre clases	Generalización	

- La mayoría de las relaciones posee cardinalidad:
  - Uno o más: $1..*$ ($1..n$)
  - Cero o más: $0..*$ ($0..n$)
  - Número fijo: m

Notas

Asociación

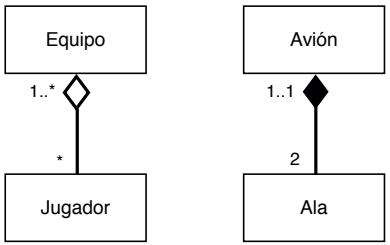


- La asociación expresa una relación (unidireccional o bidireccional) entre los objetos instanciados a partir de las clases conectadas.
- El sentido en que se recorre la asociación se denomina *navegabilidad* de la asociación.

Notas

Agregación y composición (1/2)

- Agregación y composición son relaciones Todo-Parte, en la que un objeto forma parte de la naturaleza de otro. A diferencia de la asociación, son relaciones asimétricas.
- Las diferencias entre agregación y composición son la fuerza de la relación. La agregación es una relación más débil que la composición. Ejemplo:



Notas

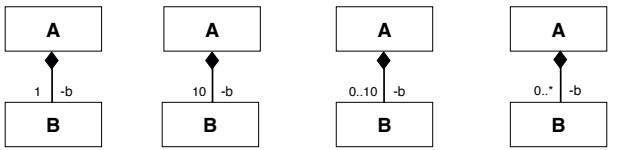
Agregación y composición (2/2)

- Si la relación es fuerte (composición), cuando se destruye el objeto contenedor también se destruyen los objetos que contiene. Ejemplo: El ala forma parte del avión y no tiene sentido fuera del mismo. Si vendemos un avión, lo hacemos incluyendo sus alas.
- En el caso de la agregación, no ocurre así. Ejemplo: Podemos vender un equipo, pero los jugadores pueden irse a otro club.
- Algunas relaciones pueden ser consideradas como agregaciones o composiciones, en función del contexto en que se utilicen. (Por ejemplo, la relación entre bicicleta y rueda).
- Algunos autores consideran que la única diferencia entre ambos conceptos radica en su implementación; así una composición sería una 'Agregación por valor'.

Notas

Implementación de la composición

- La composición es la única relación que usaremos en las prácticas de Programación 2.




```
class A {
   private:
      B b;
   ...
};
```

```
class A {
   private:
      B b[10];
   ...
};
```

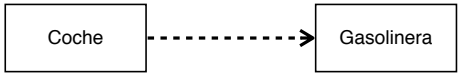
```
class A {
   private:
      vector<B> b;
      static const int N=10;
   ...
};
```

```
class A {
   private:
      vector<B> b;
   ...
};
```

Notas

Uso

- El uso es una relación no persistente (tras la misma, se termina todo contacto entre los objetos).
- Una clase A usa una clase B cuando:
  - Usa algún método de la clase B.
  - Utiliza alguna instancia de la clase B como parámetro de alguno de sus métodos.
  - Accede a sus variables privadas (esto sólo se puede hacer si son clases *amigas*).

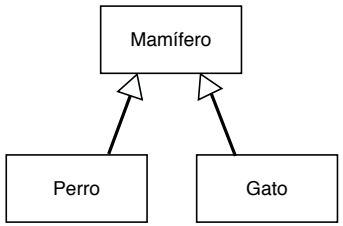



```
float Coche::Repostar(Gasolinera &g, float litros)
{
     float importe=g.dispensarGaso(litros, tipoC);
     lgaso= lgaso+litros;
     return importe;
}
```

Notas

Generalización (herencia)

- La herencia permite definir una nueva clase a partir de otra.
- Se aplica cuando hay suficientes similitudes y la mayoría de las características de la clase existente son adecuadas para la nueva clase.



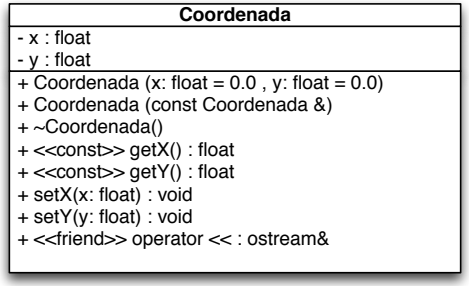
- Las *subclases* Perro y Gato heredan los métodos y atributos especificados por la *superclase* Mamífero.
- La herencia nos permite adoptar características ya implementadas por otras clases.

Notas

Ejercicios (1/3)

Ejercicio 1

Implementa la clase del siguiente diagrama:



Debes crear los ficheros `Coordenada.cc` y `Coordenada.h`, y un `makefile` para compilarlos con un programa `principal.cc`. En el main() se debe pedir al usuario dos números y crear con ellos una coordenada para imprimirla con el operador salida en el formato `x,y`. Escribe el código necesario para que cada método sea utilizado al menos una vez.

Notas

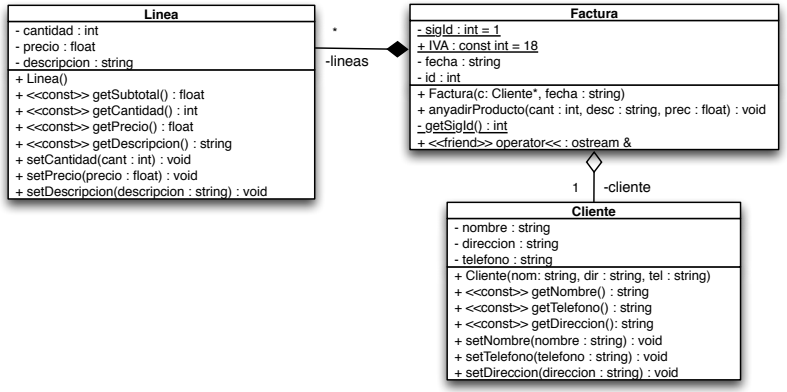
Tema 7

Introducción
Conceptos
POO en C++
Sintaxis
Funciones inline
Accesores
Forma canónica
Constructores
Asignación
Constructores de copia
Destructores
static
this
const
friend
E/S
Relaciones
Asociación
Agregación y composición
Generalización
Ejercicios

Ejercicios (2/3)

Ejercicio 2

Implementa el código correspondiente al diagrama:



Notas

Ejercicios (3/3)

Ejercicio 2 (sigue)

Se debe hacer un programa que cree una nueva factura, añada un producto y lo imprima. Desde el constructor de Factura debe llamarse al método `getSigid`, que debe devolver el valor de `sigid` e incrementarlo. Ejemplo de salida al imprimir una factura:

```
Factura nº: 12345
Fecha: 18/4/2011

Datos del cliente
-----------------------
Nombre: Agapito Piedralisa
Dirección: c/ Río Seco, 2
Teléfono: 123456789

Detalle de la factura
---------------------
Línea;Producto;Cantidad;Precio ud.;Precio total
--
1;Ratón USB;1;8,43;8,43
2;Memoria RAM 2GB;2;21,15;42,3
3;Altavoces;1;12,66;12,66

Subtotal: 63,39 €
IVA (18%): 11,41 €
TOTAL: 74.8002 €
```

Notas

Notas