@prog2ua

Tema 5: Introducción a la programación orientada a objetos

Programación 2

Grado en Ingeniería Informática
Universidad de Alicante
Curso 2022-2023

Índice

Introducción

Definición

- La *programación orientada a objetos* (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 (*procedimental*)
- Cambia el enfoque a la hora de diseñar los programas...
- ... ¡pero todo lo que has aprendido hasta ahora te sigue valiendo!

Clases y objetos (1/4)

- 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
```

- Una *clase* (o tipo compuesto) es un modelo para crear objetos de esa clase
- Un *objeto* de una determinada clase se denomina una *instancia* de la clase
- En el ejemplo anterior, `s` es una instancia/objeto de la clase `string`
- Las clases son similares a los tipos simples, aunque permiten muchas más funcionalidades

Clases y objetos (2/4)

- Un registro o `struct` es un tipo simple
- Se puede considerar como una clase “ligera” que sólo almacena datos visibles desde fuera:

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

Clases y objetos (3/4)

- Una clase contiene datos y una serie de funciones que manipulan esos datos, llamadas *funciones miembro* o *métodos*
- Se puede controlar qué datos/métodos son visibles (`public`) y cuáles están ocultos (`private`)
- Las funciones miembro pueden acceder a los datos públicos y privados de su clase
- Clase "cutre" equivalente a `struct Fecha`:*

```
class Fecha{
  public: // Datos públicos
    int dia;
    int mes;
    int anyo;
};
```

*Decimos que es "cutre" porque no ofrece ninguna ventaja con respecto a `struct Fecha`

Clases y objetos (4/4)

- Acceso directo a elementos del objeto, como en un registro:

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

- En un buen diseño orientado a objetos, normalmente no se accede directamente a los datos: para modificar los datos se usan métodos
- En el ejemplo anterior, `f.dia=100` no daría error
- Con métodos podemos controlar qué valores se dan a los datos:

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

Conceptos básicos

Conceptos básicos

- Principios en los que se basa el diseño orientado a objetos:
  - Abstracción
  - Encapsulación
  - Modularidad
  - Herencia
  - Polimorfismo

Abstracción

- La *abstracción* denota las características esenciales de un objeto y su comportamiento
- Cada objeto puede realizar tareas, informar y cambiar su estado, comunicándose 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

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 (pública) para el resto de los objetos: conjunto de métodos y datos de los cuales disponemos para comunicarnos con un objeto
- Cada objeto oculta su implementación (cómo lo hace) y expone una interfaz (qué hace)
- La encapsulación protege a las propiedades de un objeto contra su modificación: solamente los propios métodos del objeto pueden acceder a su estado

Modularidad (1/2)

- Se denomina *modularidad* a la propiedad que permite subdividir una aplicación en partes más pequeñas (*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

Modularidad (2/2)

- Una clase `miClase` se implementaría con dos ficheros fuente:
  - `miClase.h`: contiene constantes que se usen en este fichero, la declaración de la clase y la de sus métodos
  - `miClase.cc`: contiene constantes que se usen en este fichero, la implementación de los métodos y puede que tipos internos que use la clase
- El programa principal (`main`) usa y comunica las clases
- Se incluirá en un fichero aparte (por ejemplo, `prog.cc`)
- Para compilar todos los módulos y obtener un único ejecutable:

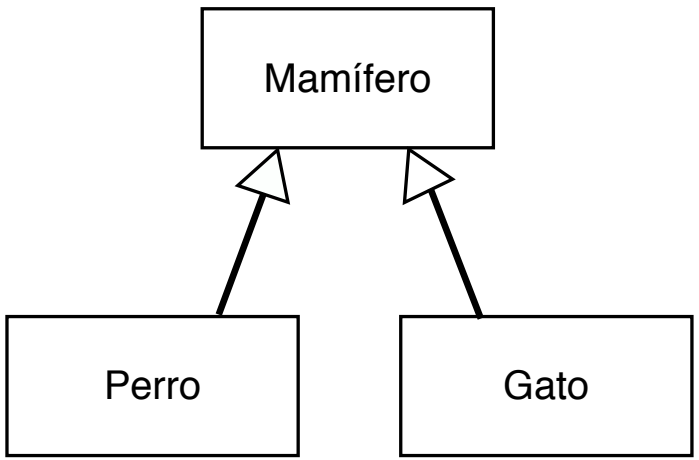
Terminal

```
$ g++ miClase1.cc miClase2.cc prog.cc -o prog
```

- Este método es adecuado sólo si tenemos pocas clases (en este ejemplo habría dos: `miClase1` y `miClase2`)
- Al final del tema veremos cómo compilar adecuadamente programas con múltiples clases utilizando la herramienta *make*

Herencia (1/2)

- No la vamos a trabajar en Programación 2
- Las clases se pueden relacionar entre sí formando una jerarquía de clasificación
- 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
- En este ejemplo, las *subclases* `Perro` y `Gato` heredan los métodos y atributos especificados por la *superclase* `Mamífero`:



Herencia (2/2)

- La herencia nos permite adoptar características ya implementadas por otras clases
- Facilita la organización de la información en diferentes niveles de abstracción
- Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen
- 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*

Polimorfismo

- No lo vamos a trabajar en Programación 2
- El *polimorfismo* es la propiedad según la cual una misma expresión hace referencia a distintas acciones
- Por ejemplo, un 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:

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

POO en C++

Declaración e implementación (1/2)

- La clase `SpaceShip` se implementará como un módulo usando dos ficheros: `SpaceShip.h` y `SpaceShip.cc`

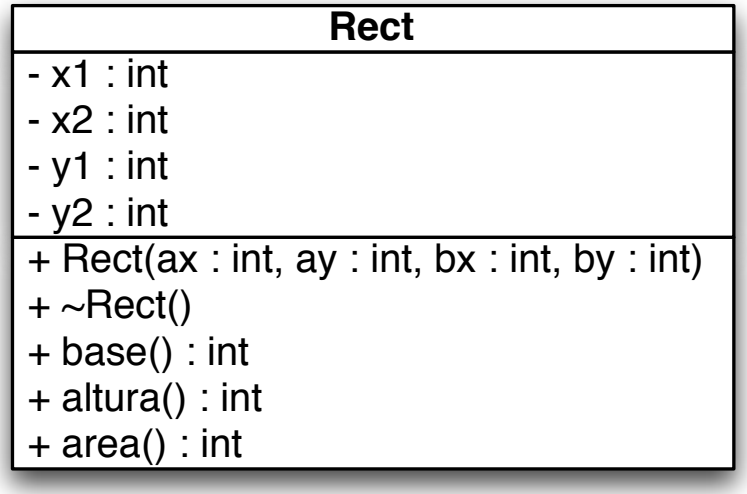
```
// SpaceShip.h (declaración de la clase)
class SpaceShip{
  private:
    int maxSpeed;
    string name;
  public:
    SpaceShip(int ms,string nm); // Constructor
    ~SpaceShip(); // Destructor
    int trip(int distance);
    string getName() const;
};
```

Declaración e implementación (2/2)

```
// SpaceShip.cc (implementación de los métodos)
#include "SpaceShip.h"

SpaceShip::SpaceShip(int ms,string nm){ // Constructor
  maxSpeed=ms;
  name=nm;
}

SpaceShip::~SpaceShip(){} // Destructor

int SpaceShip::trip(int distance){
  return distance/maxSpeed;
}

string SpaceShip::getName() const{
  return name;
}
```

Diagrama UML (1/3)

- Un *diagrama UML* permiten describir las clases y relaciones entre clases en un diseño orientado a objetos:



- El – delante de un atributo o método indica que es privado
- El + indica que es un atributo o método público
- La línea horizontal separa los atributos (parte superior) de los métodos (parte inferior)

Diagrama UML (2/3)

- Traducción a código del diagrama UML anterior:

```
// Rect.h (declaración 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();
};
```

Diagrama UML (3/3)

```
// Rect.cc (implementación de los métodos)
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 (programa principal)
int main(){
  Rect r(10,20,40,50);
  cout << r.area() << endl;
}
```

Accesores

- No es conveniente acceder directamente a los datos miembro de una clase (principio de encapsulación)
- Lo normal es definirlos como `private` y acceder a ellos implementando métodos *set/get/is* (llamados *accesores*):

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

Forma canónica

- Todas las clases deben implementar estos cuatro métodos:
  - Constructor
  - Destructor
  - Constructor de copia
  - Operador de asignación
- Si alguno no ha sido definido en la clase, el compilador lo crea por defecto

Constructor (1/7)

- El *constructor* se invoca automáticamente cuando se crea un objeto de la clase
- Las clases deben tener al menos un método constructor
- Si no definimos un constructor, el compilador creará uno por defecto sin parámetros (los datos miembros de los objetos creados así estarán sin inicializar)
- Una clase puede tener varios constructores con parámetros distintos (el constructor puede *sobrecargarse*)
- La sobrecarga es un tipo de polimorfismo

Constructor (2/7)

- Ejemplos de constructor:

```
Fecha::Fecha(){ // Sin parámetros
  dia=1;
  mes=1;
  anyo=1900;
}

Fecha::Fecha(int d,int m,int a){ // Con tres parámetros
  dia=d;
  mes=m;
  anyo=a;
}
```

- Llamadas al constructor:

```
Fecha f;
Fecha f(10,2,2010);
Fecha f(); // ¡Error de compilación!
```

Constructor (3/7)

- Los constructores (al igual que otras funciones) pueden tener parámetros por defecto
- Estos valores por defecto sólo se ponen en el fichero de cabecera (`.h`):

```
// Fecha.h
class Fecha{
  ...
  Fecha(int d=1,int m=1,int a=1900);
  ...
}
```

- Con este constructor podríamos crear objetos de varias formas:

```
Fecha f; // dia = 1, mes = 1, anyo = 1900
Fecha f(10,2,2010); // dia = 10, mes = 2, anyo = 2010
Fecha f(10); // dia = 10, mes = 1, anyo = 1900
Fecha f(18,5); // dia = 18,  mes = 5, anyo = 1900
```

Constructor (4/7)

- Los parámetros por defecto del ejemplo anterior se mostrarían de la siguiente manera en un diagrama UML:

Fecha
- dia: int - mes: int - anyo: int
+ Fecha (dia: int=1, mes: int=1, anyo: int=1900) ...

Constructor (5/7)

- Si los parámetros que se le pasan al constructor son incorrectos no debería de crearse el objeto
- Esto se puede controlar mediante el uso de *excepciones*:
  - Podemos lanzar una excepción con `throw` para indicar que se ha producido un error
  - Podemos capturar una excepción con `try/catch` para reaccionar ante el error
- Si se produce una excepción y no la capturamos, el programa terminará inmediatamente
- Las excepciones sólo deben usarse cuando no hay otra opción (por ejemplo, en los constructores)

Constructor (6/7)

- Ejemplo de uso de excepciones:

```
int root(int n){
  if(n<0)
    throw exception(); // Lanza la excepción y termina
  return sqrt(n);
}

int main(){
  try{ // Intentamos ejecutar estas instrucciones
    int result=root(-1); // Provoca una excepción
    cout << result << endl; // Esta línea no se ejecuta
  }
  catch(...){ // Si hay una excepción la capturamos aquí
    cout << "Negative number" <<  endl;
  }
}
```

Constructor (7/7)

- Ejemplo de constructor con excepción:

```
Coordenada::Coordenada(int cx,int cy){
  if(cx>=0 && cy>=0){
    x=cx;
    y=cy;
 }
 else
   throw exception();
}
```

```
int main(){
  try{
    Coordenada c(-2,4); // Este objeto no llega a crearse
  }
  catch(...){
    cout << "Coordenada incorrecta" << endl;
  }
}
```

Destructor (1/2)

- El *destructor* de la clase debe liberar los recursos (normalmente memoria dinámica) que el objeto esté usando
- Una clase sólo tiene una función destructor que no tiene argumentos y no devuelve ningún valor
- Es un método con igual nombre que la clase y precedido por el carácter ~:

```
// Declaración
~Fecha();
// Implementación
Fecha::~Fecha(){
  // Liberar la memoria reservada (si fuera necesario)
}
```

Destructor (2/2)

- Todas las clases necesitan un destructor y si no se especifica, el compilador crea uno por defecto
- El compilador llama automáticamente al destructor del objeto cuando acaba su ámbito
- También se invoca al destructor al hacer `delete`
- El destructor de un objeto invoca implícitamente a los destructores de todos sus atributos

Constructor de copia (1/2)

- Un *constructor de copia* crea un objeto a partir de otro objeto existente:

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

// Implementación
Fecha::Fecha(const Fecha &f){
  dia=f.dia;
  mes=f.mes;
  anyo=f.anyo;
}
```

Constructor 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); // Constructor de copia
Fecha f2=f1; // Constructor de copia
f2=f1; // Aquí no se invoca al constructor sino a =
```

  - Un objeto se pasa por valor a una función:

```
void funcion(Fecha f);
funcion(f);
```

- Si no se especifica ningún constructor de copia, el compilador crea uno por defecto que hace una copia atributo a atributo del objeto

Operador de asignación

- No lo vamos a ver en Programación 2
- El *operador de asignación* (=) permite una asignación directa de dos objetos:

```
Fecha f1(10,2,2011); // Constructor
Fecha f2; // Constructor
f2=f1; // Operador de asignación
```

- Por defecto, el compilador crea un operador de asignación que copia atributo a atributo
- Podemos redefinirlo para nuestras clases si lo consideramos necesario

Declaraciones *inline* (1/2)

- Los métodos con poco código se pueden implementar directamente en la declaración de la clase (declaración *inline*):

```
// Rect.h
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();
};
```

Declaraciones *inline* (2/2)

- Es más eficiente declarar funciones *inline*
- 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)
- Las funciones *inline* también se pueden implementar fuera de la declaración de clase, en el fichero `.cc`, usando la palabra reservada `inline`:

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

Métodos constantes (1/2)

- Los métodos que no modifican los atributos del objeto se pueden declarar como *métodos constantes*:

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

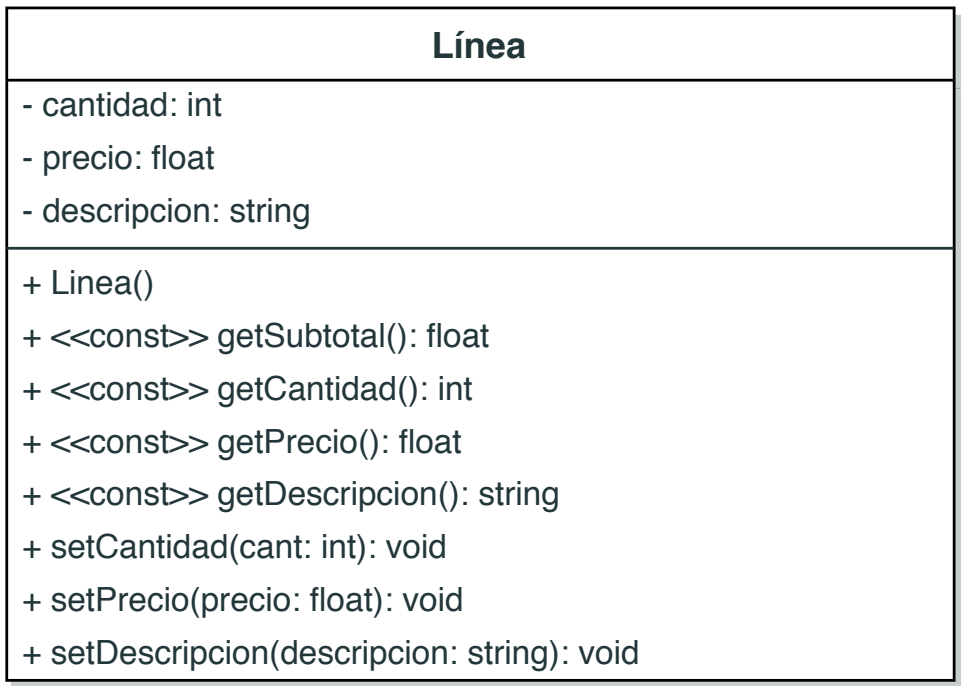
- En un objeto constante sólo se pueden invocar métodos constantes:

```
int Fecha::getDia(){ // No se ha declarado como const
  return dia;
}
int main(){
  const Fecha f(10,10,2011);
  cout << f.getDia() << endl; // Error de compilación
}
```

- Los métodos `get` deben declararse constantes, ya que se limitan a devolver valores y no modifican nunca al objeto

Métodos constantes (2/2)

- Se representan poniendo `<<const>>` delante del nombre del método en los diagramas UML
- En este ejemplo hay cuatro métodos constantes (`getSubtotal`, `getCantidad`, `getPrecio` y `getDescription`):



Funciones amigas

- Una *función amiga* no pertenece a la clase pero puede acceder a su parte privada
- Se declara usando la palabra reservada `friend` en su declaración:

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

```
void unaFuncionAmiga(int x,MiClase &c){
  c.datoPrivado=x; // Correcto, porque es amiga
}
```

Sobrecarga de la entrada/salida (1/4)

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

```
Fecha f;
cin >> f;
cout << f;
```

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

```
friend ostream& operator<<(ostream &o,const Fecha &f);
friend istream& operator>>(istream &o,Fecha &f);
```

Sobrecarga de la entrada/salida (2/4)

- Declaración:

```
class Fecha{
  friend ostream& operator<<(ostream &os,const Fecha &f);
  friend istream& operator>>(istream &is,Fecha &f);
  public:
    Fecha(int dia=1,int mes=1,int anyo=1900);
    ...
  private:
    int dia,mes,anyo;
};
```

Sobrecarga de la entrada/salida (3/4)

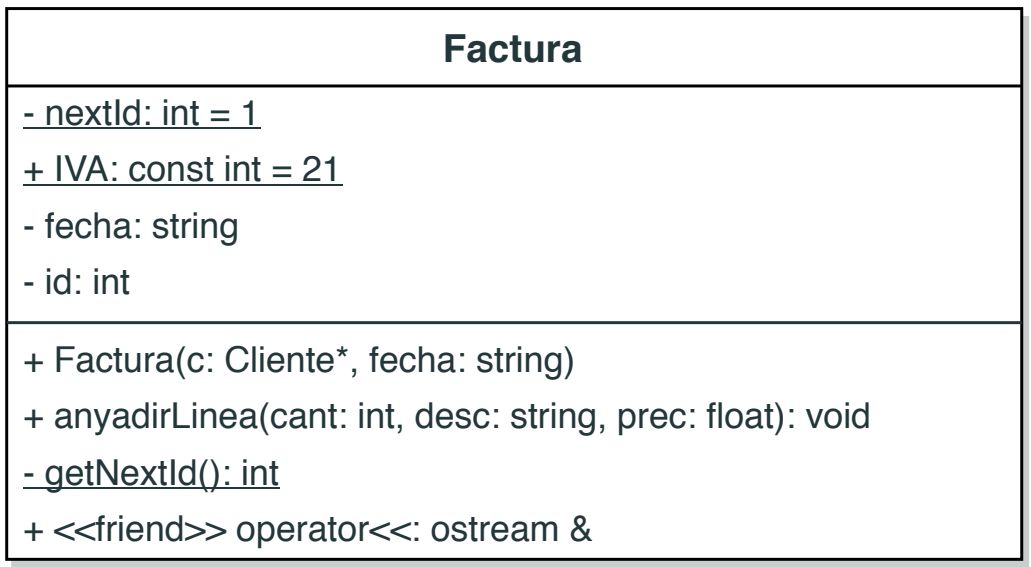
- Implementación:

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

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

Sobrecarga de la entrada/salida (4/4)

- En un diagrama UML se pondrá la palabra `<<friend>>` delante del operador, ya que se trata de una función amiga
- En este ejemplo, la clase tiene sobrecargado el operador de salida (`operator<<`):



Atributos y métodos de clase (1/4)

- 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* producen la misma salida para todos los objetos de la clase y sólo pueden acceder a atributos de clase
- También se llaman atributos y métodos *estáticos*
- Se declaran mediante la palabra reservada `static` al definir la clase:

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

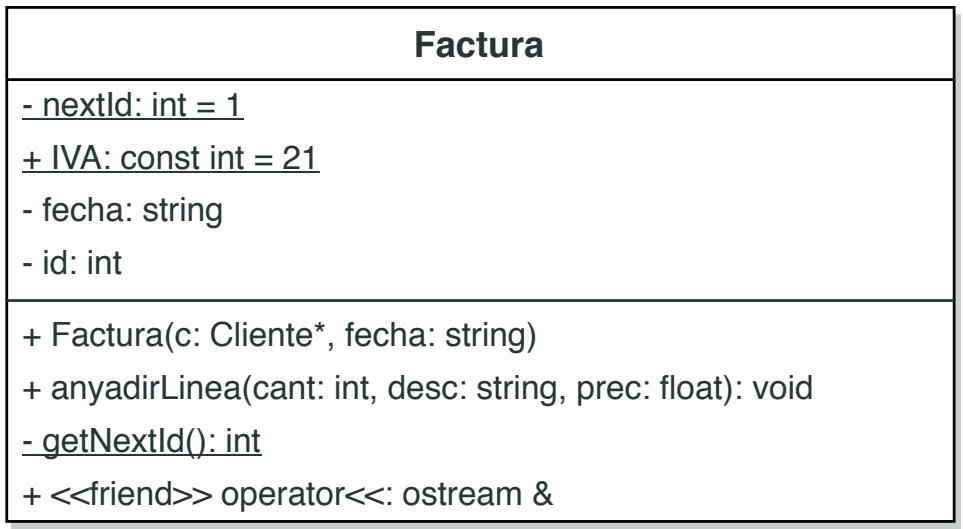
Atributos y métodos de clase (2/4)

- El el fichero donde se implementan los métodos (`.cc`) no se debe poner la palabra `static` al definirlos
- Para la clase `Fecha` del ejemplo anterior, tendríamos el siguiente código:

```
// Fecha.cc
string Fecha::getFormato(){ // No se pone static
   ...
}

bool Fecha::setFormato(string s){ // No se pone static
   ...
}
```

Atributos y métodos de clase (3/4)

- Se representan subrayados en los diagramas UML
- En este ejemplo hay dos atributos estáticos (`IVA` y `nextId`) y un método estático (`getNextId`):



Atributos y métodos de clase (4/4)

- Si 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
class Fecha{
  ...
  static const string finMundo;
  ...
};
```

```
// Fecha.cc
const string Fecha::finMundo="2020"; // No se pone static
```

- Acceso a atributos o métodos estáticos desde fuera de la clase:

```
cout << Fecha::diasPorAnyo << endl; // Atributo estático
cout << Fecha::getFormato() << endl; // Método estático
```

El puntero `this`

- El puntero `this` es una pseudovariable que no se declara ni se puede modificar
- Es un argumento implícito que reciben todos los métodos (excluyendo los estáticos) y que apunta al objeto receptor del mensaje
- 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; Ojo: le asigna a dia su propio valor
  this->dia=dia;
  cout << this->dia << endl;
}
```

Objetos y gestión de memoria

Objetos automáticos y objetos dinámicos (1/5)

- Teniendo en cuenta su permanencia en la memoria, los objetos pueden ser *automáticos* o *dinámicos*
- Los objetos *dinámicos* se crean en tiempo de ejecución utilizando el operador `new`
- Permanecen en memoria (generalmente en *heap*) hasta que se eliminan explícitamente a través del operador `delete`*
- Los objetos *automáticos* se crean (automáticamente) en la memoria (generalmente en *stack*) en tiempo de ejecución cuando se entra en su ámbito
- Se destruyen (automáticamente) cuando se sale de dicho ámbito

*Todos estos conceptos fueron descritos en el Tema 4

Objetos automáticos y objetos dinámicos (2/5)

- Ejemplo de creación de un objeto automático y de uno dinámico:

```
void func(){
  Fecha f1; // Objeto automático
  Fecha *f2=NULL;
  f2=new Fecha; // Objeto dinámico
}
int main(){
  func();
  ...
}
```

- Al terminar la función `func` el objeto `f1` ya no está en memoria, pero el objeto apuntado por `f2` sí, aunque ya no es accesible
- Problema: el espacio de memoria al que apunta `f2` no se borra y ocupa memoria hasta que el programa acabe

Objetos automáticos y objetos dinámicos (3/5)

- En el ejemplo anterior, podemos devolver el puntero para que el objeto apuntado por `f2` se pueda seguir usando:

```
Fecha* func(){ // func devuelve un puntero a Fecha
  Fecha f1; // Objeto automático
  f1.setDia(10);
  Fecha *f2=NULL;
  f2=new Fecha; // Objeto dinámico
  f2->setDia(20);
  return f2;
}
int main(){
  Fecha *f=NULL;
  f=func();
  cout << f->getDia(); // Muestra "20"
  f->setMes(1);
  delete f; // Objeto dinámico borrado de memoria
}
```

Objetos automáticos y objetos dinámicos (4/5)

- Así quedaría el ejemplo anterior si devolviéramos un objeto automático en lugar de uno dinámico:

```
Fecha func(){
  Fecha f1; // Objeto automático
  f1.setDia(10);
  return f1; // Llama al constructor de copia
}
int main(){
  Fecha f=func(); // Objeto automático
  cout << f.getDia(); // Muestra "10"
  f.setDia(1);
}
```

- El constructor de copia será invocado en este caso para inicializar el objeto `f` del `main`

Objetos automáticos y objetos dinámicos (5/5)

- Asignación de objetos automáticos y dinámicos:

```
Fecha f1,f2,*f3=new Fecha,*f4=new Fecha;
f2.setDia(15);
f3->setDia(10);
f4->setDia(20);
f1=f2; // Se llama al operador de asignación
Fecha f5=f1; // Llamada al constructor de copia
f3=f4; // El objeto asignado originalmente a f3 ya
       // no se puede borrar
       // f3 y f4 apuntan ahora al mismo objeto
cout << f3->getDia(); // Muestra "20"
delete f3;
f4->setDia(5); // Igual de incorrecto que f3->setDia(5);
               // f3 y f4 apuntan a memoria no válida
```

- La última línea es incorrecta, pero podría llegar a funcionar
- Al eliminar un objeto se marcan sus posiciones de memoria para ser reasignadas (no se pone justo en ese momento la memoria a cero o se reasigna a nuevos datos)

Copia profunda y copia superficial

- Dada una clase `A` con un campo `B` que es un objeto dinámico:

```
class A{B *b; ... }
```

- Decimos que el constructor de copia de `A` realiza una *copia profunda* del objeto si reserva nueva memoria para él:

```
A::A(const A &a){
  ...
  b=new B(a.b);
}
```

- Decimos que el constructor de copia de `A` realiza una *copia superficial* del objeto si hace que el nuevo puntero apunte a donde el anterior:

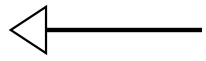
```
A::A(const A &a){
  ...
  b=a.b;
}
```

- Las dos opciones pueden ser convenientes según el problema

Relaciones

Relaciones entre objetos

- Principales tipos de relaciones entre objetos y clases:

Entre objetos	Asociación	
	Agregación	
	Composición	
	Uso	
Entre clases	Generalización	

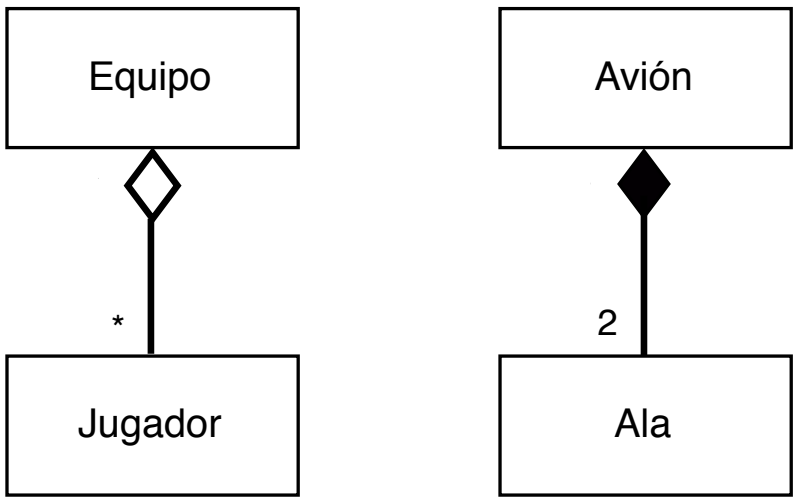
- La mayoría de las relaciones posee cardinalidad:
  - Uno o más: 1..∗ (1..*n*)
  - Cero o más: ∗
  - Número fijo: *m*
- En Programación 2 vamos a trabajar solo la agregación y la composición

Agregación y composición (1/6)

- *Agregación* y *composición* son relaciones todo-parte en las que un objeto forma parte de la naturaleza de otro
- Son relaciones asimétricas
- La diferencia entre agregación y composición es la fuerza de la relación: la agregación es una relación más débil que la composición

Agregación y composición (2/6)

- En la composición, cuando se destruye el objeto contenedor también se destruyen los objetos que contiene
  - Ej: 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
  - Ej: podemos vender un equipo, pero los jugadores pueden irse a otro club (no desaparecen con el equipo)

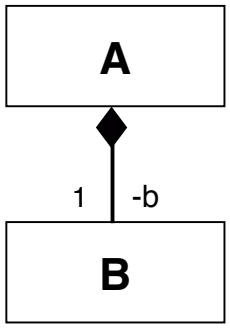


Agregación y composición (3/6)

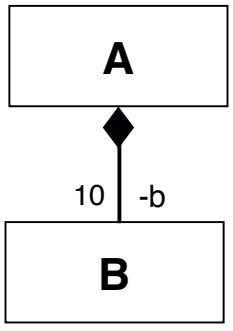
- Algunas relaciones pueden ser consideradas como agregaciones o composiciones en función del contexto en que se utilicen
  - Ej: la relación entre bicicleta y rueda
- Algunos autores consideran que la única diferencia entre ambos conceptos radica en su implementación: una composición sería una “agregación por valor”

Agregación y composición (4/6)

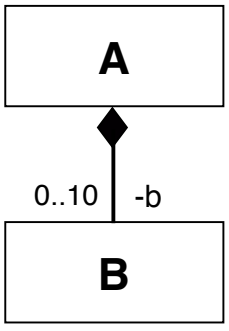
- Implementación de la composición:



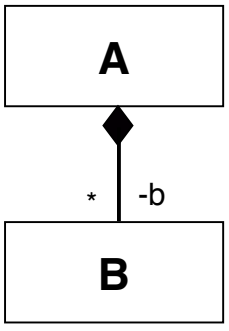

```
class A {
   private:
      B b;
   ...
};
```




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



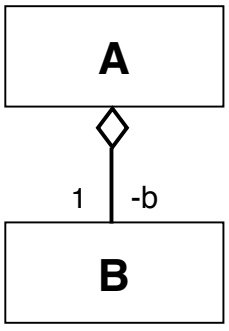

```
class A {
   private:
      vector<B> b;
      static const int N=10;
   ...
};
```



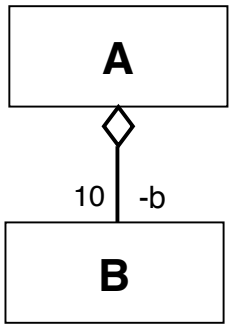

```
class A {
   private:
      vector<B> b;
   ...
};
```

Agregación y composición (5/6)

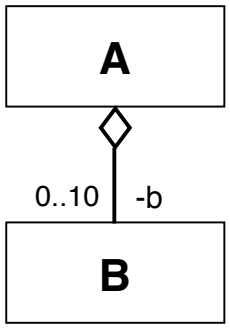
- Implementación de la agregración:



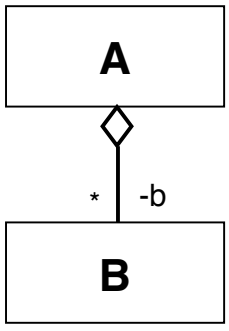

```
class A {
   private:
      B *b;
   ...
};
```




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




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




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

Agregación y composición (6/6)

- Ejemplo de implementación de la agregración:

```
class A{
  private:
    B *b;
  public:
    A(B *b){ this->b=b; } // Constructor
};
```

```
int main(){ // Dos formas de implementar la agregación
  // 1- Mediante un puntero
  B *b=new B;
  A a(b); // Llamada al constructor
  // 2- Mediante un objeto
  B b;
  A a(&b); // Llamada al constructor
}
```

Compilación

El proceso de compilación

- La tarea de traducir un programa fuente en ejecutable se realiza en dos fases:
  - *Compilación*: el compilador traduce un programa fuente en un programa en código objeto (no ejecutable)
  - *Enlace*: el enlazador (*linker*) junta el programa en código objeto con las librerías del lenguaje (C/C++) y genera el ejecutable
- En C++ se realizan las dos fases con la siguiente instrucción:

Terminal

```
$ g++ programa.cc -o programa
```

- Con la opción `-c` sólo se compila, generando código objeto (`.o`), pero sin hacer el enlace:

Terminal

```
$ g++ programa.cc -c
```

Compilación separada (1/2)

- Cuando un programa se compone de varios ficheros fuente (`.cc`), lo que debe hacerse para obtener el ejecutable es:
  1. Compilar cada fuente por separado, obteniendo varios ficheros en código objeto (`.o`):

     Terminal

     ```
     $ g++ -c C1.cc
     $ g++ -c C2.cc
     $ g++ -c prog.cc -c
     ```

  2. Enlazar los ficheros en código objeto con las librerías del lenguaje y generar un ejecutable:

     Terminal

     ```
     $ g++ C1.o C2.o prog.o -o prog
     ```

- Si tiene pocos ficheros fuente, se puede hacer todo de una vez:

  Terminal

  ```
  $ g++ C1.cc C2.cc prog.cc -o prog
  ```

Compilación separada (2/2)

- Problema: tenemos un fichero de cabecera `.h` que se usa en varios ficheros fuente `.cc`
- ¿Qué hay que hacer si se cambia algo en el `.h`?
  - Opción 1: lo recompilo todo (a lo “bestia”)
  - Opción 2: busco “a mano” dónde se usa y sólo recompilo esas clases
  - Opción 3: busco automáticamente dónde se usa y sólo recompilo esas clases
- La mejor es la “Opción 3” y hay un programa llamado *make* que nos ayuda a hacerlo

La herramienta *make* (1/6)

- La herramienta *make* ayuda a compilar programas grandes
- Permite establecer *dependencias* entre ficheros
- Compila un fichero cuando alguno de los ficheros de los que depende cambia
- El fichero de texto *makefile* especifica las dependencias entre los ficheros y qué hacer cuando algo cambia

La herramienta *make* (2/6)

- La herramienta *make* busca por defecto un fichero llamado *makefile*
- En este fichero se describe un *objetivo* principal (normalmente el programa ejecutable) y una serie de objetivos secundarios
- El formato de cada objetivo del fichero *makefile* es:

```
<objetivo> : <dependencias>
[tabulador]<instrucción>
```

- El algoritmo del programa *make* es muy sencillo: *"Si la fecha de alguna dependencia es más reciente que la del objetivo, ejecutar instrucción"*

La herramienta *make* (3/6)

- Imaginemos que tenemos los siguientes ficheros:

```
// C1.cc
#include "C1.h"
...
```

```
// C2.cc
#include "C2.h"
#include "C1.h"
...
```

```
// prog.cc
#include "C1.h"
#include "C2.h"
...
int main(){
...
}
```

La herramienta *make* (4/6)

- El fichero `makefile` sería:*

```
prog : C1.o C2.o prog.o
    g++ -Wall -g C1.o C2.o prog.o -o prog
C1.o : C1.cc C1.h
    g++ -Wall -g -c C1.cc
C2.o : C2.cc C2.h C1.h
    g++ -Wall -g -c C2.cc
prog.o : prog.cc C1.h C2.h
    g++ -Wall -g -c prog.cc
```

*La opción `-Wall` muestra todos los *warnings* y `-g` añade información para el depurador

La herramienta *make* (5/6)

- En el ejemplo anterior, si se modifica `C2.cc` y se ejecuta *make*:

Terminal

```
$ make
g++ -Wall -g -c C2.cc
g++ -Wall -g C1.o C2.o prog.o -o prog
```

- Y si se modifica `C2.h` y se ejecuta *make*:

Terminal

```
$ make
g++ -Wall -g -c C2.cc
g++ -Wall -g -c prog.cc
g++ -Wall -g C1.o C2.o prog.o -o prog
```

La herramienta *make* (6/6)

- Ejemplo anterior usando constantes (más “profesional”):*

```
CC = g++
CFLAGS = -Wall -g
OBJS = C1.o C2.o prog.o

prog : $(OBJS)
    $(CC) $(CFLAGS) $(OBJS) -o prog
C1.o : C1.cc C1.h
    $(CC) $(CFLAGS) -c C1.cc
C2.o : C2.cc C2.h C1.h
    $(CC) $(CFLAGS) -c C2.cc
prog.o : prog.cc C1.h C2.h
    $(CC) $(CFLAGS) -c prog.cc
clean:
    rm -rf $(OBJS)
```

*Más información en: http://es.wikipedia.org/wiki/Make

Directivas del preprocesador

- Se pueden producir errores de compilación cuando un fichero de cabecera se incluye en múltiples ficheros de nuestro código
- El compilador piensa que se está declarando múltiples veces la clase que hay en ese fichero de cabecera
- Hay que utilizar las instrucciones `#ifndef`, `#define` y `#endif` en nuestros ficheros de cabecera para evitarlo

```
// C1.h
#ifndef _C1_H_
#define _C1_H_
...
class C1{
  ...
};
#endif
```

Ejercicios

Ejercicios (1/3)

Ejercicio 1

Implementa la clase del siguiente diagrama:

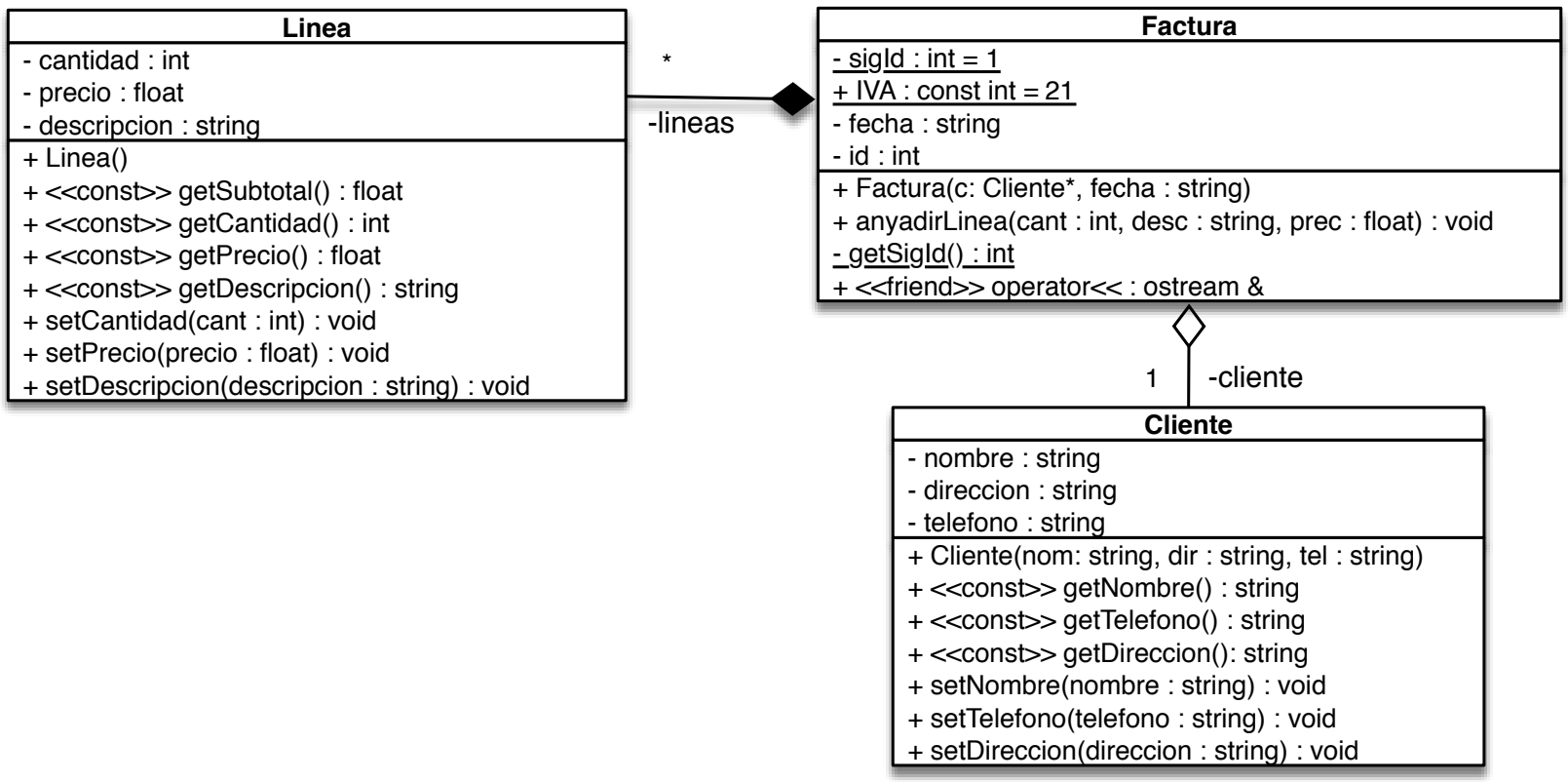
Coordenada
- x : float - y : float
+ Coordenada (cx: float=0, cy: float = 0) + Coordenada (const Coordenada &) + ~Coordenada() + <<const>> getX() : float + <<const>> getY() : float + setX (cx:float) : void + setY (cy:float) : void + <<friend>> operator << : ostream &

Debes crear los ficheros `Coordenada.cc` y `Coordenada.h`, además de un *makefile* para compilarlos junto 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.

Ejercicios (2/3)

Ejercicio 2

Implementa el código correspondiente al siguiente diagrama UML:



Ejercicios (3/3)

Ejercicio 2 (continuación)

Se debe hacer un programa que cree una nueva factura, añada un producto y lo imprima. Desde el constructor de `Factura` se llamará al método `getSigId`, que devolverá el valor de `sigId` y lo incrementará. 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 (21%): 13.3119 €
TOTAL: 76.7019 €
```