

Tema 5: Introducció a la programació orientada a objectes

Programació 2

Grau en Enginyeria Informàtica
Universitat d'Alacant
Curs 2025-2026



1. Introducció
2. Conceptes bàsics
3. POO en C++
4. Objectes i gestió de memòria
5. Relacions
6. Compilació
7. Exercicis

Introducció

- La *programació orientada a objectes* (POO) és un paradigma de programació que usa objectes i les seues interaccions per a dissenyar aplicacions i programes informàtics
- L'aplicació sencera es redueix a un conjunt d'objectes i les seues relacions
- C++ és un llenguatge orientat a objectes, encara que també permet programació imperativa (*procedural*)
- Canvia l'enfocament a l'hora de dissenyar els programes...
- ... però tot el que has après fins ara et continua sent útil!

Classes i objectes (1/4)

- En Programació 2 ja hem usat classes i objectes:

```
int i; // Declarem una variable i de tipus int  
string s; // Declarem un objecte s de classe string
```

- Una *classe* (o tipus compost) és un model per a crear objectes d'aquesta classe
- Un *objecte* d'una determinada classe es denomina una *instància* de la classe
- En l'exemple anterior, *s* és una instància/objecte de la classe `string`
- Les classes són similars als tipus simples, encara que permeten moltes més funcionalitats

Classes i objectes (2/4)

- Un registre o `struct` és un tipus simple
- Es pot considerar com una classe “lleugera” que només emmagatzema dades visibles des de fora:

```
struct Data{  
    int dia;  
    int mes;  
    int any;  
};
```

Classes i objectes (3/4)

- Una classe conté dades i una sèrie de funcions que manipulen aquestes dades, anomenades *funciones membre* o *mètodes*
- Es pot controlar quines dades/mètodes són visibles (`public`) i quins estan ocults (`private`)
- Les funcions membre poden accedir a les dades públiques i privades de la seua classe
- Classe “cutre” equivalent a `struct Data`:*

```
class Data{  
    public: // Dades públiques  
        int dia;  
        int mes;  
        int any;  
};
```

*Diem que és “cutre” perquè no ofereix cap avantatge respecte a `struct Data`

Classes i objectes (4/4)

- Accés directe a elements de l'objecte, com en un registre:

```
Data d;  
d.dia=12;
```

- En un bon disseny orientat a objectes, normalment no s'accedeix directament a les dades: per a modificar-les s'usen mètodes
- En l'exemple anterior, `d.dia=100` no donaria error
- Amb mètodes podem controlar quins valors es donen a les dades:

```
class Data{  
    private: // Només accessible des de mètodes de la  
            classe  
        int dia;  
        int mes;  
        int any;  
    public:  
        bool setData(int d, int m, int a){...};  
};
```


Conceptes bàsics

- Principis en què es basa el disseny orientat a objectes:
 - Abstracció
 - Encapsulació
 - Modularitat
 - Herència
 - Polimorfisme

- L'*abstracció* denota les característiques essencials d'un objecte i el seu comportament
- Cada objecte pot fer tasques, informar i canviar el seu estat, comunicant-se amb altres objectes en el sistema sense revelar com s'implementen aquestes característiques
- El procés d'abstracció permet seleccionar les característiques rellevants dins d'un conjunt i identificar comportaments comuns per a definir noves classes
- El procés d'abstracció té lloc en la fase de disseny

- L'*encapsulació* significa reunir tots els elements que poden considerar-se pertanyents a una mateixa entitat al mateix nivell d'abstracció
- La *interfície* és la part de l'objecte que és visible (pública) per a la resta dels objectes: conjunt de mètodes i dades dels quals disposem per a comunicar-nos amb un objecte
- Cada objecte oculta la seua implementació (com ho fa) i exposa una interfície (què fa)
- L'encapsulació protegeix les propietats d'un objecte contra la seua modificació: només els mètodes de l'objecte poden accedir-ne a l'estat

- Es denomina *modularitat* la propietat que permet subdividir una aplicació en parts més xicotetes (*mòduls*) tan independents com siga possible
- Aquests mòduls es poden compilar per separat, però tenen connexions amb altres mòduls
- Generalment, cada classe s'implementa en un mòdul independent, encara que classes amb funcionalitats similars també poden compartir mòdul

Modularitat (2/2)

- Una classe `laMeuaClasse` s'implementaria amb dos fitxers font:
 - `laMeuaClasse.h`: conté constants que s'usen en aquest fitxer, la declaració de la classe i la dels seus mètodes
 - `laMeuaClasse.cc`: conté constants que s'usen en aquest fitxer, la implementació dels mètodes i potser tipus interns
- El programa principal (`main`) usa i comunica les classes
- S'inclourà en un fitxer a banda (per exemple, `prog.cc`)
- Per a compilar tots els mòduls i obtenir un únic executable:

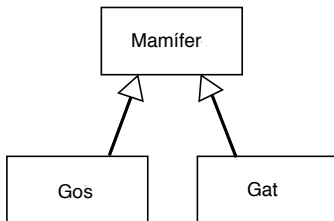
Terminal

```
$ g++ lameuaclasses1.cc lameuaclasses2.cc prog.cc  
-o prog
```

- Aquest mètode és adequat només si tenim poques classes (en aquest exemple n'hi hauria dos: `laMeuaClasse1` i `laMeuaClasse2`)
- Al final del tema veurem com compilar adequadament programes amb múltiples classes utilitzant l'eina *make*

Herència (1/2)

- No la treballarem en Programació 2
- Les classes es poden relacionar entre si formant una jerarquia de classificació
- L'*herència* permet definir una nova classe a partir d'una altra
- S'aplica quan hi ha suficients similituds i la majoria de les característiques de la classe existent són adequades per a la nova classe
- En aquest exemple, les *subclasse*s Perro i Gato hereten els mètodes i atributs especificats per la *superclasse* Mamífero:



- L'herència ens permet adoptar característiques ja implementades per altres classes
- Facilita l'organització de la informació en diferents nivells d'abstracció
- Els objectes hereten les propietats i el comportament de totes les classes a les quals pertanyen
- Els objectes derivats poden compartir (i estendre) el comportament sense haver de tornar a implementar-ho
- Quan un objecte hereta de més d'una classe es diu que hi ha *herència múltiple*

- No ho treballarem en Programació 2
- El *polimorfisme* és la propietat segons la qual una mateixa expressió fa referència a diferents accions
- Per exemple, un mètode `desplaçar` pot referir-se a accions diferents si es tracta d'un avió o d'un cotxe
- Comportaments diferents, associats a objectes diferents, poden compartir el mateix nom
- Les referències i les col·leccions d'objectes poden contindre objectes de diferents tipus:

```
Mamifer *a=new Gos;  
Mamifer *b=new Gat;  
Mamifer *c=new Gavina;
```

POO en C++

Declaració i implementació (1/2)

- La classe `SpaceShip` s'implementarà com un mòdul usant dos fitxers: `SpaceShip.h` i `Spaceship.cc`

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

Declaració i implementació (2/2)

```
// Spaceship.cc (implementació dels mètodes)
#include "SpaceShip.h"

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

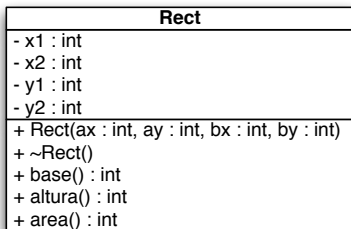
SpaceShip::~SpaceShip(){} // Destructor

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

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

Diagrama UML (1/3)

- Un *diagrama UML* permet descriure les classes i relacions entre classes en un disseny orientat a objectes:



- El - davant d'un atribut o mètode indica que és privat
- El + indica que és un atribut o mètode públic
- La línia horitzontal separa els atributs (part superior) dels mètodes (part inferior)

Diagrama UML (2/3)

- Traducció a codi del diagrama UML anterior:

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

Diagrama UML (3/3)

```
// Rect.cc (implementació dels mètodes)
Rect::Rect(int ax,int ai,int bx,int by){
    x1=ax;
    y1=ai;
    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;
}
```

Accessors

- No és convenient accedir directament a les dades membre d'una classe (principi d'encapsulació)
- Allò normal és definir-los com `private` i accedir-hi implementant mètodes *set/get/is* (anomenats *accesors*):

Fecha
- dia : int - mes : int - anyo : int
+ getDia () : int + getMes () : int + getAnyo() : int + setDia (d : int) : void + setMes (m : int) : void + setAnyo (a : int) : void + isBisiesto () : bool

- Els accesors `set` ens permeten controlar que els valors dels atributs siguin correctes

- Totes les classes han d'implementar aquests quatre mètodes:
 - Constructor
 - Destructor
 - Constructor de còpia
 - Operador d'assignació
- Si algun d'ells no ha estat definit en la classe, el compilador el crea per defecte

Constructor (1/7)

- El *constructor* s'invoca automàticament quan es crea un objecte de la classe
- Les classes han de tindre almenys un mètode constructor
- Si no definim un constructor, el compilador crearà un per defecte sense paràmetres (les dades membres dels objectes creats així estaran sense inicialitzar)
- Una classe pot tindre diversos constructors amb paràmetres diferents (el constructor pot *sobrecarregar-se*)
- La sobrecàrrega és un tipus de polimorfisme

Constructor (2/7)

- Exemples de constructor:

```
Data::Data() { // Sense paràmetres
    dia=1;
    mes=1;
    any=1900;
}

Data::Data(int d,int m,int a){ // Amb tres paràmetres
    dia=d;
    mes=m;
    any=a;
}
```

- Crides al constructor:

```
Data d;
Data d(10,2,2010);
Data d(); // Error de compilació!
```

Constructor (3/7)

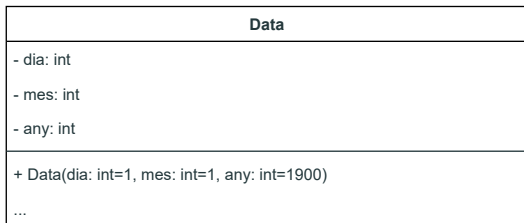
- Els constructors (com altres funcions) poden tindre paràmetres per defecte
- Aquests valors per defecte només es posen en el fitxer de capçalera (.h):

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

- Amb aquest constructor podríem crear objectes de diverses formes:

```
Data d; // dia = 1, mes = 1, any = 1900
Data d(10,2,2010); // dia = 10, mes = 2, any = 2010
Data d(10); // dia = 10, mes = 1, any = 1900
Data d(18,5); // dia = 18, mes = 5, any = 1900
```

- Els paràmetres per defecte de l'exemple anterior es mostrarien de la següent manera en un diagrama UML:



- Si els paràmetres que rep el constructor són incorrectes no hauria de crear-se l'objecte
- Això es pot controlar mitjançant l'ús d'*excepcions* :
 - Podem llançar una excepció amb `throw` per indicar que s'ha produït un error
 - Podem capturar una excepció amb `try/catch` per a reaccionar davant l'error
- Si es produeix una excepció i no la capturem, el programa acabarà immediatament
- Les excepcions només cal usar-les quan no hi ha una altra opció (per exemple, en els constructors)

Constructor (6/7)

- Exemple d'ús d'excepcions:

```
int root(int n){  
    if(n<0)  
        throw exception(); // Llança l'excepció i acaba  
    return sqrt(n);  
}  
  
int main(){  
    try{ // Intentem executar aquestes instruccions  
        int result=root(-1); // Provoca una excepció  
        cout << result << endl; // Aquesta línia no s'executa  
    }  
    catch(...){ // Si hi ha una excepció la capturem aquí  
        cout << "Negative number" << endl;  
    }  
}
```

Constructor (7/7)

- Exemple de constructor amb excepció:

```
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); // Aquest objecte no es crea  
    }  
    catch(...){  
        cout << "Coordenada incorrecta" << endl;  
    }  
}
```


Destructor (1/2)

- El *destructor* de la classe ha d'alliberar els recursos (normalment memòria dinàmica) que l'objecte estiga usant
- Una classe només té una funció destructor que no té arguments i no retorna cap valor
- És un mètode amb el mateix nom que la classe i precedit pel caràcter ~:

```
// Declaració
~Data();

// Implementació
Data::~Data() {
    // Alliberar la memòria reservada (si fóra necessari)
}
```

Destructor (2/2)

- Totes les classes necessiten un destructor i si no s'especifica, el compilador en crea un per defecte
- El compilador crida automàticament el destructor de l'objecte quan n'acaba l'àmbit
- També s'invoca el destructor en fer `delete`
- El destructor d'un objecte invoca implícitament els destructors de tots els seus atributs

Constructor de còpia (1/2)

- Un *constructor de còpia* crea un objecte a partir d'un altre objecte existent:

```
// Declaració
Data(const Data &f);

// Implementació
Data::Data(const Data &d) {
    dia=d.dia;
    mes=d.mes;
    any=d.any;
}
```

Constructor de còpia (2/2)

- El constructor de còpia s'invoca automàticament quan:
 - Una funció retorna un objecte
 - S'inicialitza un objecte quan es declara:

```
Data d2(d1); // Constructor de còpia  
Data d2=d1; // Constructor de còpia  
d2=d1; // Ací no s'invoca el constructor sinó =
```

- Un objecte es passa per valor a una funció:

```
void func(Data d);  
func(d);
```

- Si no s'especifica cap constructor de còpia, el compilador en crea un per defecte que fa una copia atribut a atribut de l'objecte

Operador d'assignació

- No ho veurem en Programació 2
- L'operador d'assignació (=) permet una assignació directa de dos objectes:

```
Data d1(10,2,2011); // Constructor  
Data d2; // Constructor  
d2=d1; // Operador d'assignació
```

- Per defecte, el compilador crea un operador d'assignació que copia atribut a atribut
- Podem redefinir-lo per a les nostres classes si ho considerem necessari

Declaracions *inline* (1/2)

- Els mètodes amb poc codi es poden implementar directament en la declaració de la classe (declaració *inline*):

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

Declaracions *inline* (2/2)

- És més eficient declarar funcions *inline*
- Quan es compila el codi generat per a les funcions *inline*, s'insereix en el punt on s'invoca la funció (en lloc de fer-ho en un altre lloc i fer una crida)
- Les funcions *inline* també es poden implementar fora de la declaració de classe, en el fitxer `.cc`, usant la paraula reservada `inline`:

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

Mètodes constants (1/2)

- Els mètodes que no modifiquen els atributs de l'objecte es poden declarar com a mètodes *constants*:

```
int Data::getDia() const{ // Mètode constant
    return dia;
}
```

- En un objecte constant només es poden invocar mètodes constants:

```
int Data::getDia(){ // No s'ha declarat com const
    return dia;
}
int main(){
    const Data d(10,10,2011);
    cout << d.getDia() << endl; // Error de compilació
}
```

- Els mètodes `get` han de declarar-se constants, ja que es limiten a retornar valors i no modifiquen mai l'objecte

Mètodes constants (2/2)

- Es representen posant `<<const>>` davant del nom del mètode en els diagrames UML
- En aquest exemple hi ha quatre mètodes constants (`getSubtotal`, `getCantidad`, `getPrecio` i `getDescripcion`):

Línea
- cantidad: int - precio: float - descripcion: string
+ Linea() + <<const>> getSubtotal(): float + <<const>> getCantidad(): int + <<const>> getPrecio(): float + <<const>> getDescripcion(): string + setCantidad(cant: int): void + setPrecio(precio: float): void + setDescripcion(descripcion: string): void

Funcions amigues

- Una *funció amiga* no pertany a la classe però pot accedir-ne a la part privada
- Es declara usant la paraula reservada `friend` en la declaració:

```
class LaMeuaClasse{  
    friend void unaFuncioAmiga(int, LaMeuaClasse &);  
    public:  
        ...  
    private:  
        int dadaPrivada;  
};
```

```
void unaFuncioAmiga(int x, LaMeuaClasse &c){  
    c.dadaPrivada=x; // Correcte, perquè és amiga  
}
```

Sobrecàrrega de l'entrada/eixida (1/4)

- Podem sobrecarregar les operacions d'entrada/eixida de qualsevol classe:

```
Data d;  
cin >> d;  
cout << d;
```

- El problema és que no poden ser funcions membre d'una classe perquè el primer operant (`cin/cout`) no és un objecte d'aquesta classe (és un `stream`)
- Els operadors se sobrecarreguen usant funcions amigues:

```
friend ostream& operator<<(ostream &o,const Data &d);  
friend istream& operator>>(istream &o,Data &d);
```

- Declaració:

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

Sobrecàrrega de l'entrada/eixida (3/4)

- Implementació:

```
ostream& operator<<(ostream &os,const Data &d){  
    os << d.dia << "/" << d.mes << "/" << d.any;  
    return os;  
}
```

```
istream& operator>>(istream &is,Data &d){  
    char dummy;  
    is >> d.dia >> dummy >> d.mes >> dummy >> d.any;  
    return is;  
}
```

Sobrecàrrega de l'entrada/eixida (4/4)

- En un diagrama UML es posarà la paraula `<<friend>>` davant de l'operador, ja que es tracta d'una funció amiga
- En aquest exemple, la classe té sobrecarregat l'operador d'eixida (`operator<<`):

Factura
<u>- nextId: int = 1</u> <u>+ IVA: const int = 21</u> - fecha: string - id: int
+ Factura(c: Cliente*, fecha: string) + anyadirLinea(cant: int, desc: string, prec: float): void <u>- getNextId(): int</u> + <<friend>> operator<<: ostream &

Atributs i mètodes de classe (1/4)

- Els *atributs de classe* tenen el mateix valor per a tots els objectes de la classe (són com a variables globals per a la classe)
- Els *mètodes de classe* produeixen la mateixa eixida per a tots els objectes de la classe i només poden accedir a atributs de classe
- També es diuen atributs i mètodes *estàtics*
- Es declaren mitjançant la paraula reservada `static` al definir la classe:

```
class Data{  
    public:  
        static const int setmanesPerAny=52;  
        static const int diesPerSetmana=7;  
        static const int diesPerAny=365;  
        static string getFormat();  
        static bool setFormat(string);  
    private:  
        static string cadenaFormat;  
};
```

Atributs i mètodes de classe (2/4)

- Al fitxer on s'implementen els mètodes no s'ha de posar la paraula `static` en definir-los
- Per a la classe `Data` de l'exemple anterior, tindríem el següent codi:

```
// Data.cc  
string Data::getFormat(){ // No posem static  
    ...  
}  
  
bool Data::setFormat(string s){ // No posem static  
    ...  
}
```


Atributs i mètodes de classe (3/4)

- Es representen subratllats en els diagrames UML
- En aquest exemple hi ha dos atributs estàtics (`IVA` i `nextId`) i un mètode estàtic (`getNextId`):

Factura
<u>- nextId: int = 1</u> <u>+ IVA: const int = 21</u> - fecha: string - id: int
+ Factura(c: Cliente*, fecha: string) + anyadirLinea(cant: int, desc: string, prec: float): void <u>- getNextId(): int</u> + <<friend>> operator<<: ostream &

Atributs i mètodes de classe (4/4)

- Si l'atribut estàtic no és un tipus simple o no és constant, ha de declarar-se en la classe però agafar el valor fora d'ella:

```
// Data.h
class Data{
    ...
    static const string fiMon;
    ...
};
```

```
// Data.cc
const string Data::fiMon="2020"; // No posem static
```

- Accés a atributs o mètodes estàtics des de fora de la classe:

```
cout << Data::diesPerAny << endl; // Atribut estàtic
cout << Data::getFormat() << endl; // Mètode estàtic
```

El punter `this`

- El punter `this` és una pseudovariàble que no es declara ni es pot modificar
- És un argument implícit que reben tots els mètodes (excloent els estàtics) i que apunta l'objecte receptor del missatge
- És necessari quan volem desambiguar el nom del paràmetre o quan volem passar com a argument l'objecte a una funció niada:

```
void Data::setDia(int dia){  
    // dia=dia; Atenció: assigna a dia el seu propi valor  
    this->dia=dia;  
    cout << this->dia << endl;  
}
```

Objectes i gestió de memòria

Objectes automàtics i objectes dinàmics (1/5)

- Quant a la permanència en la memòria, els objectes poden ser *automàtics* o *dinàmics*
- Els objectes dinàmics es creen en temps d'execució utilitzant l'operador `new`
- Romanen vius en la memòria (generalment al *heap*) fins que s'eliminen explícitament a través de l'operador `delete`*
- Els objectes automàtics es reserven (automàticament) en la memòria (generalment al *stack*) en temps d'execució quan s'entra en el seu àmbit
- Es destrueixen (automàticament) quan s'ix d'aquest àmbit

*Tots aquests conceptes van ser descrits en el Tema 4

Objectes automàtics i objectes dinàmics (2/5)

- Creació d'un objecte automàtic i un dinàmic:

```
void func() {  
    Data d1; // Objecte automàtic  
    Data *d2=NULL;  
    d2=new Data; // Object dinàmic  
}  
int main(){  
    func();  
    ...  
}
```

- Després de cridar `func`, l'objecte `d1` ja no està en memòria però l'objecte apuntat per `d2` si, encara que ja no és accessible
- Problema: l'espai de memòria apuntat per `d2` no serà esborrat i romandrà fins que el programa acabe

Objectes automàtics i objectes dinàmics (3/5)

- En l'exemple anterior podem retornar el punter perquè l'objecte apuntat originalment per d2 es pugui seguir usant:

```
Data* func(){ // func retorna un punter a Data
    Data d1; // Objecte automàtic
    d1.setDia(10);
    Data *d2=NULL;
    d2=new Data; // Objecte dinàmic
    d2->setDia(20);
    return d2;
}
int main(){
    Data *d=NULL;
    d=func();
    cout << d->getDia(); // Mostra "20"
    d->setMes(1);
    delete d; // Objecte dinàmic esborrat de memòria
}
```

Objectes automàtics i objectes dinàmics (4/5)

- Aquest és el codi si retornem un objecte automàtic en lloc d'un objecte dinàmic:

```
Data func() {  
    Data d1; // Objecte automàtic  
    d1.setDia(10);  
    return d1; // Crida al constructor de còpia  
}  
  
int main(){  
    Date d=func(); // Objecte automàtic  
    cout << d.getDia(); // Mostra "10"  
    d.setDia(1);  
}
```

- El constructor de còpia serà invocat en aquest cas per inicialitzar l'objecte d al main

Objectes automàtics i objectes dinàmics (5/5)

- Assignació d'objectes automàtics o dinàmics:

```
Data d1,d2,*d3= new Data,*d4= new Data;  
d2.setDia(15);  
d3->setDia(10);  
d4->setDia(20);  
d1=d2; // Es crida a l'operador d'assignació  
Data d5=d1; // Crida al constructor de còpia  
d3=d4; // L'objecte assignat originalment a d3 ja  
        // no es pot borrar  
        // d3 i d4 apunten ara al mateix objecte  
cout << d3->getDia(); // Mostra "20"  
delete d3;  
d4->setDia(5); // Tan incorrecte com d3->setDia(5);  
               // d3 i d4 apunten a memòria no vàlida
```

- L'última línia és incorrecta, pero podria funcionar
- En eliminar un objecte es marquen les seues posicions de memòria per a ser reassignades (no es posa just en eixe moment la memòria a zero o es reassigna a noves dades)

Còpia profunda i còpia superficial

- Considereu un objecte de classe A amb un camp que és un objecte dinàmic:

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

- El constructor de còpia d'A realitza una *còpia profunda* de l'objecte si reserva noves posicions de memòria:

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

- El constructor de còpia d'A realitza una *còpia superficial* de l'objecte si fa que el nou punter apunte al objecte ja existent:

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

- Les dues opcions poden ser convenientes segons el context.

Relacions

Relacions entre objectes

- Principals tipus de relacions entre objectes i classes:

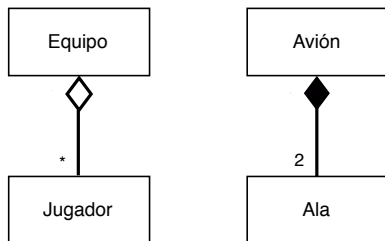
Entre objectes	Asociación	————
	Agregación	◊————
	Composició	◆————
	Uso	←-----
Entre classes	Generalización	◁————

- La majoria de les relacions posseeixen cardinalitat:
 - Un o més: $1..*$ ($1..n$)
 - Zero o més: $*$
 - Número fix: m
- En Programació 2 treballarem només l'agregació i la composició

- *Agregació* i *composició* són relacions tot-part en les quals un objecte forma part de la naturalesa d'un altre
- Són relacions asimètriques
- La diferència entre agregació i composició és la força de la relació: l'agregació és una relació més feble que la composició

Agregació i composició (2/6)

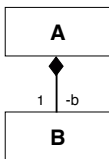
- En la composició, quan es destrueix l'objecte contenidor també es destrueixen els objectes que conté
 - Ej: l'ala forma part de l'avió i no té sentit fora del mateix (si venem un avió, el fem incloent-ne les ales)
- En el cas de l'agregació, no ocorre així
 - Ej: podem vendre un equip, però els jugadors poden anar-se'n a un altre club (no desapareixen amb l'equip)



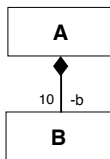
- Algunes relacions poden ser considerades com a agregacions o composicions en funció del context en què s'utilitzen
 - Ej: la relació entre bicicleta i roda
- Alguns autors consideren que l'única diferència entre tots dos conceptes radica en la implementació: una composició seria una “agregació per valor”

Agregació i composició (4/6)

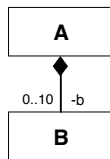
- Implementació de la composició:



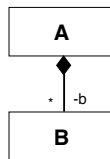
```
class A {
    private:
        B b;
    ...
};
```



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



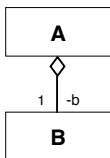
```
class A {
    private:
        vector<B> b;
        static const int N=10;
    ...
};
```



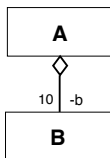
```
class A {
    private:
        vector<B> b;
    ...
};
```


Agregació i composició (5/6)

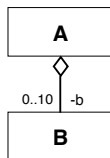
- Implementació de la agregació:



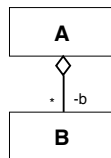
```
class A {  
    private:  
        B *b;  
    ...  
};
```



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



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



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

Agregació i composició (6/6)

- Exemple d'implementació de la agregació:

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

```
int main(){ // Dues maneres d'implementar l'agregació  
    // 1- Mitjançant un punter  
    B b=new B;  
    A a(b); // Crida al constructor  
    // 2- Mitjançant un objecte  
    B b;  
    A a(&b); // Crida al constructor  
}
```

Compilació

El procés de compilació

- La tasca de traduir un programa font en executable es realitza en dues fases:
 - *Compilació*: el compilador tradueix un programa font en un programa en codi objecte (no executable)
 - *Enllaç*: l'enllaçador (*linker*) junta el programa en codi objecte amb les llibreries del llenguatge (C/C++) i genera l'executable
- En C++ es realitzen les dues fases amb la següent instrucció:

Terminal

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

- Amb l'opció `-c` només es compila, generant codi objecte (`.o`), però sense fer l'enllaç:

Terminal

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

Compilació separada (1/2)

- Quan un programa es compon de diversos fitxers font (.cc), allò que cal fer per a obtenir l'executable és:

1. Compilar cada font per separat, obtenint diversos fitxers en codi objecte (.o):

Terminal

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

2. Enllaçar els fitxers en codi objecte amb les llibreries del llenguatge i generar un executable:

Terminal

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

- Si té pocs fitxers font, es pot fer tot d'una vegada:

Terminal

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

- Problema: tenim un fitxer de capçalera `.h` que s'usa en diversos fitxers font `.cc`
- Què cal fer si es canvia alguna cosa en el `.h`?
 - Opció 1: ho recompila tot (a tort i a dret)
 - Opció 2: busque “a mà” on s'usa i només recompila aquestes classes
 - Opció 3: busque automàticament on s'usa i només recompila aquestes classes
- La millor és l'“Opció 3” i hi ha un programa anomenat *make* que ens ajuda a fer-ho

- L'eina *make* ajuda a compilar programes grans
- Permet establir *dependències* entre fitxers
- Compila un fitxer quan algun dels fitxers de què depén canvia
- El fitxer de text *makefile* especifica les dependències entre els fitxers i què fer quan alguna cosa canvia

L'eina *make* (2/6)

- L'eina *make* busca per defecte un fitxer anomenat *makefile*
- En aquest fitxer es descriu un *objectiu* principal (normalment el programa executable) i una sèrie d'objectius secundaris
- El format de cada objectiu del fitxer *makefile* és:

```
<objectiu> : <dependències>  
[tabulador]<instrucció>
```

- L'algorisme del programa *make* és molt senzill: “*Si la data d'alguna dependència és més recent que la de l'objectiu, executa la instrucció*”

L'eina *make* (3/6)

- Imaginem que tenim els següents fitxers:

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

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

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

L'eina *make* (4/6)

- El fitxer `makefile` seria:*

```
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
```

*L'opció `-Wall` mostra tots els *warnings* i `-g` afegirà informació per al depurador

L'eina *make* (5/6)

- En l'exemple anterior, si es modifica `C2.cc` i s'executa *make*:

Terminal

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

- I si es modifica `C2.h` i s'executa *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
```

L'eina *make* (6/6)

- Exemple anterior usant constants (més “professional”):*

```
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)
```

*Més informació en: <http://es.wikipedia.org/wiki/Make>

Directives del preprocesador

- Es poden produir errors de compilació quan un fitxer de capçalera s'inclou en múltiples fitxers del nostre codi
- El compilador pensa que s'està declarant múltiples vegades la classe que hi ha en eixe fitxer de capçalera
- Cal utilitzar les instruccions `#ifndef`, `#define` i `#endif` en els nostres fitxers de capçalera per a evitar-lo

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

Exercicis

Exercicis (1/3)

Exercici 1

Implementeu la classe del següent diagrama:

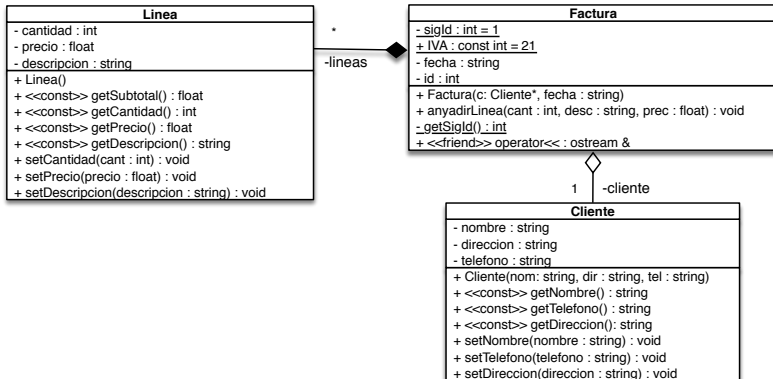
Coordenada
- x : float - y : float
+ Coordenada (cx: float=0, cy: float = 0) + Coordenada (const Coordenada &) + ~Coordenada() + <<const>> getX() : float + <<const>> getY() : float + setX (cx:float) : void + setY (cy:float) : void + <<friend>> operator << : ostream &

Heu de crear els fitxers `Coordenada.cc` i `Coordenada.h`, a més d'un *makefile* per a compilar-los juntament amb un programa principal.cc. En el main cal demanar l'usuari dos números i crear amb ells una coordenada per a imprimir-la amb l'operador eixida en el format (x, y) . Escriviu el codi necessari perquè cada mètode siga usat almenys una vegada.

Exercicis (2/3)

Exercici 2

Implementeu el codi corresponent al següent diagrama UML:



Exercicis (3/3)

Exercici 2 (continuació)

Heu de fer un programa que cree una nova factura, hi afegisca un producte i la imprimisca. Des del constructor de `Factura` es cridarà el mètode `getSigId`, que retornarà el valor de `sigId` i l'incrementarà. Exemple d'eixida en imprimir una factura:

```
Factura núm.: 12345
Data: 18/4/2011

Dades del client
-----
Nom: Agapito Piedralisa
Adreça: c/ Riu Sec, 2
Telèfon: 123456789

Detall de la factura
-----
Línia;Producte;Quantitat;Preu un.;Preu total
--
1;Ratolí USB;1;8.43;8.43
2;Memòria RAM 2GB;2;21.15;42.3
3;Altaveus;1;12.66;12.66

Subtotal: 63.39 €
IVA (18%): 13.3119 €
TOTAL: 76.7019 €
```