Programmazione Avanzata per il Calcolo Scientifico
Lezione N. 6

Luca Formaggia

MOX
Dipartimento di Matematica "F. Brioschi"
Politecnico di Milano

A.A. 2006/2007

L'ereditarietà

- Classe Base e Classe derivata
- Sovrascittura (overriding) di metodi
- Un test
- Costruttori e distruttori di classi derivate
- Classe astratta
- Casting dinamico
- Ereditarietà multipla

Aggregazione e delegazione

- Modalità di contenimento
- Delegazione con ereditarietà privata

Come non spararsi nel piede?

Ereditarietà

Una classe rappresenta un *concetto* e fornisce delle *funzionalità*.
L'ereditarietà è un meccanismo che permette:

- Estendere la funzionalità di una classe creando una classe *derivata*
- Esprimere concetti comuni, sviluppando una *gerarchia* di classi.
- Implementare il polimorfismo: esprimere un concetto generico (astrazione) le cui funzionalità specifiche sono determinate dalla particolare istanza.

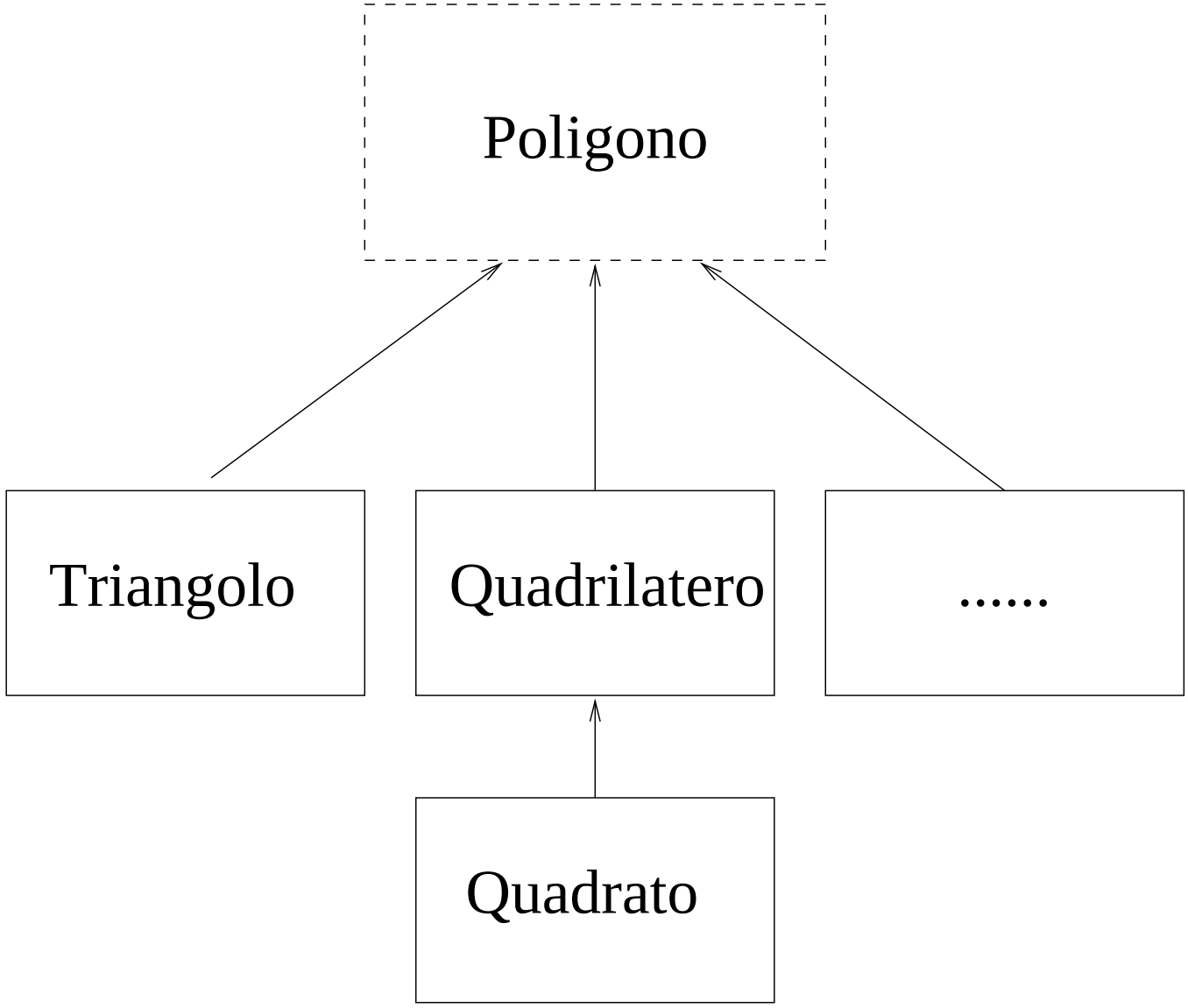
Esempio

Il concetto di poligono è un concetto astratto, su cui possiamo definire funzionalità (metodi) diverse: `area()`, `diameter()`, `convexHull()`, `inscribedRadius()` etc.

Queste vengono poi *concretizzate* dalla istanza che rappresenta un *poligono* specifico , per esempio un Quadrato.

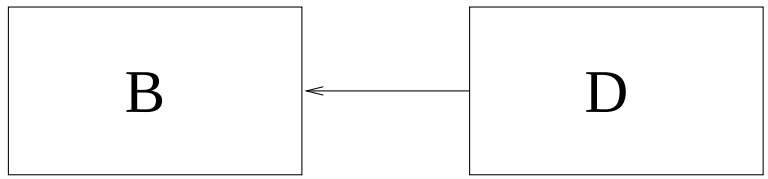
È il classico esempio di una *gerarchia* di concetti che in C++ si esprime con la ereditarietà pubblica.

Gerarchia concettuale



Base e Derivata

La dizione *classe base* e *classe derivata* indica due classi in un relazione gerarchica tra di loro. Si dice che la classe `D` deriva da `B`.



La relazione si dice polimorfa se le funzionalità della classe derivata sono un 'superset'di quelli dalla classe base. Si dice anche che in questo caso `D` is-a `B` (relazione *is-a* o *works-like-a*).

N.B.: La relazione *is-a* è solo una delle relazioni possibili tra classi. L'ereditarietà pubblica va utilizzata solo per tale tipo di relazione.

La relazione è-un (is-a)

La relazione è-un è una relazione molto forte tra classi.
Permette di utilizzare una referenza (o un puntatore) alla classe base su un oggetto della classe derivata. Perchè questo non causi problemi occorre che la classe derivata non fonrisca di meno ne richieda di più di quanto dichiarato nella interfaccia pubblica della classe base.

PROMISE NO LESS, REQUIRE NO MORE (H. Sutter)

Estratto da Polygon.hpp

```
class Polygon {
public:
explicit Polygon(Vertices v, bool convex=false);
virtual ~Polygon();
bool isConvex() const;
virtual double diameter() const;
// qui vanno le altre funzionalita‘..
protected:
bool isconvex;
};

class Quadrilateral: public Polygon {
public:
explicit Quadrilateral(Vertices const & v);
virtual ~Quadrilateral();
virtual double diameter() const;
// qui vanno le altre funzionalita‘.. };
```

```
class Square: public Quadrilateral {
public:
explicit Square(Vertices const & v);
...
// Metodi propri del quadrato
double diagLen() const; };
```

Il meccanismo qui illustrato è quello della *ereditarietà pubblica*, espressa dalla keyword `public` nella dichiarazione di dipendenza della classe: `class D: public B`.

La classe `Square` dipende *direttamente* e pubblicamente da `Quadrilateral` e *indirettamente* da `Polygon`.

Il meccanismo di accesso attivato dalla ereditarietà pubblica `B` $\leftarrow$ `D` è il seguente:

1. i membri (variabili e metodi) pubblici e protetti (protected) di `B` sono accessibili direttamente da `D`. I membri protetti sono privati per tutte le classi che non derivano pubblicamente (direttamente o indirettamente) da `B`. I membri privati di `B` sono accessibili solo da `B`.
2. Un puntatore o una referenza a una classe derivata si convertono automaticamente in puntatori alla classe base.
3. Metodi ridefiniti in `D` nascondono metodi con lo stesso nome definiti in `B` (come nei namespace annidati).
4. metodi dichiarati virtual nella classe `B` vengono sovrascritti (overriding) da metodi con la stessa signature in `D`.

overriding (sovrascrittura)

```
void f (Polygon & p){
...
p.diameter();
...
int main() {
Square s; ...
f(s);
```

Alla funzione `f` passo come argomento attuale uno `Square`. Dato che è un passaggio per referenza ciò è possibile per la regola 2. Ma la cosa più importante è che `d.diameter()` nel corpo della funzione usa il metodo definito da Square e non quello in `Polygon`, in quanto `diameter()` è un *metodo virtuale* della classe base.

L'overriding richiede puntatori o referenze

```
void f (Polygon * p){
...
p->diameter();
```

Si ottiene lo stesso effetto di prima.

```
void f (Polygon p){
...
p.diameter();
```

Nel secondo caso, invece, la funzione accetta come argomenti attuali solo oggetti di tipo `Polygon`!
Si dice che una referenza o un puntatore è usato in modo polimorfico se è usato per indirizzare un oggetto di una classe derivata.

Un test sull'overriding/shading

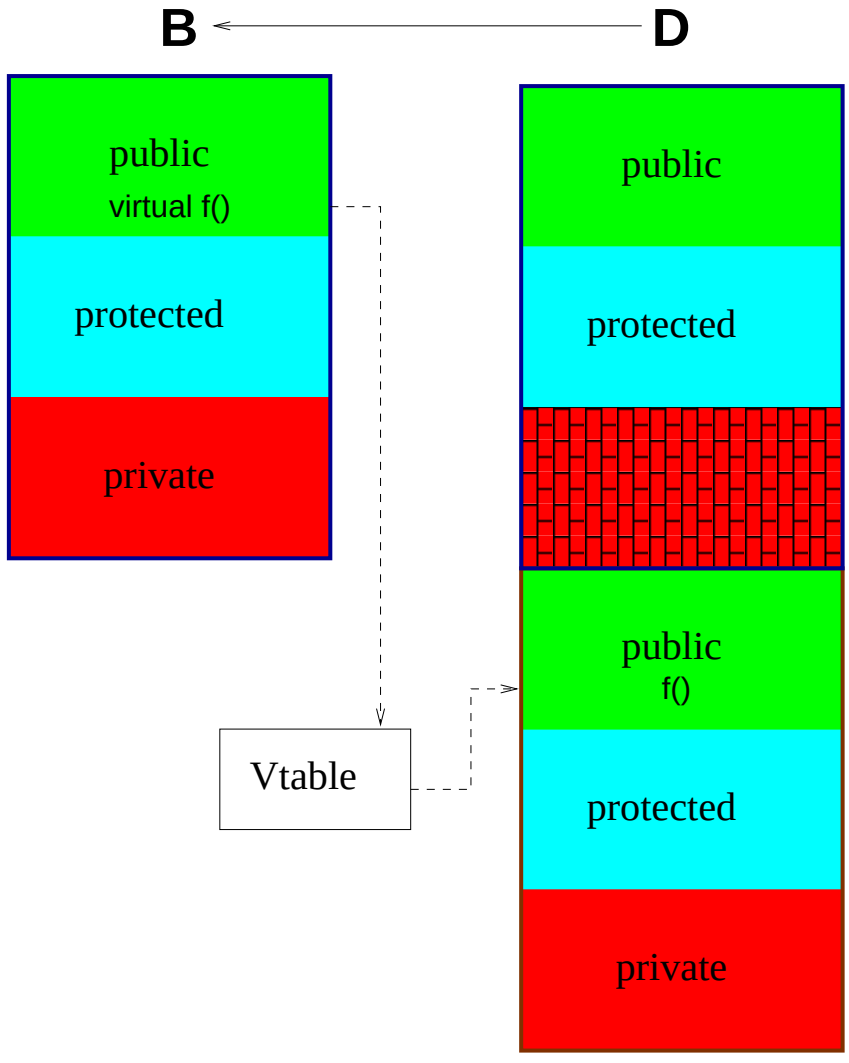
```
class B{
public:
void f1(){cout«”B::f1”«endl;}
virtual void f2(){cout«” B::f2”«endl;}
};

class D: public B {
public:
void f1(){cout«”D::f1”«endl;}
virtual void f2(){cout«” D::f2”«endl;}
};
```

Cosa appare sullo schermo?

```
D d; B b; D * dp=new D;
B & br(d); B * bp(&d);
b.f1();   B::f1
b.f2();   B::f2
d.f1();   D::f1
d.f2();   D::f2
dp->f1(); D::f1
dp->f2(); D::f2
br.f1(); B::f1
br.f2(); D::f2
bp->f1(); B::f1
bp->f2(); D::f2
d.B::f1(); B::f1
```

Il meccanismo della ereditarietà pubblica



La costruzione di una classe derivata

La costruzione di un oggetto di una classe derivata avviene nel modo seguente:

- Vengono costruiti (ricorsivamente) i membri della classe base usando o il costruttore di default della classe base o quello fornito nella lista di inizializzazione della classe derivata.
- Vengono costruiti i membri propri della classe derivata (eventualmente con il costruttore di default).

Esempio:

```
Quadrilateral::Quadrilateral(Vertices const & v):Polygon(v)
{// codice per deteminare la convessita‘ ;}
```

Conseguenza di questo algoritmo è che i membri ereditati dalla classe base sono disponibili per la costruzione di membri propri della classe derivata.

Distruttore della classe derivata

La distruzione di un oggetto di una classe derivata avviene nell'ordine inverso della costruzione:

- Si distruggono (usando eventualmente il distruttore di default) i membri della classe derivata.
- Si distruggono (ricorsivamente) i membri della(e) classe(i) base.

Il distruttore di una classe base (ereditarietà pubblica) deve sempre essere dichiarato *virtual* quando la classe derivata introduce nuove variabili rispetto alla classe base (per sicurezza fatelo sempre).

```
Polygon * p=new Square(v);
delete p; //Devo lanciare il distruttore di Square!
```

Classe astratta

Il concetto di Poligono è in realtà un concetto astratto che serve a modello per le sue implementazioni concrete (Quadrato, Triangolo, etc.). Non avrebbe quindi senso istanziare un oggetto di tipo `Polygon`. Come si può esprimere questo fatto in C++?

Il C++ introduce il concetto di *classe astratta*, che è una classe base di una gerarchia di classi in cui (almeno) un metodo è posto eguale al puntatore nullo (si ricorda che una funzione/metodo è di fatto un puntatore).

Esempio di una classe astratta

```
class Polygon {
public:
explicit Polygon(Vertices v, bool convex=false);
virtual ~Polygon();
bool isConvex() const;
virtual double diameter() const=0;
// qui vanno le altre funzionalita'..
protected:
bool isconvex;
};
```

L'aver dato valore nullo a un metodo (che quindi dovrà necessariamente essere sovrascritto da una classe derivata) impedisce l'istanziazione:

```
Polygon p;//NO errore di compilazione
```

dynamic_cast<T>

Il comando `dynamic_cast<T>` può essere usato per verificare a quale classe derivata punta effettivamente un puntatore alla classe base. Infatti

```
B* b=new D;
D* dp=dynamic_cast<D *>(b);
```

ritorna in `dp` un puntatore a `D` se `D` è derivata pubblicamente da `B`. Altrimenti ritorna il puntatore nullo.
Si può usare anche con le *referenze*.

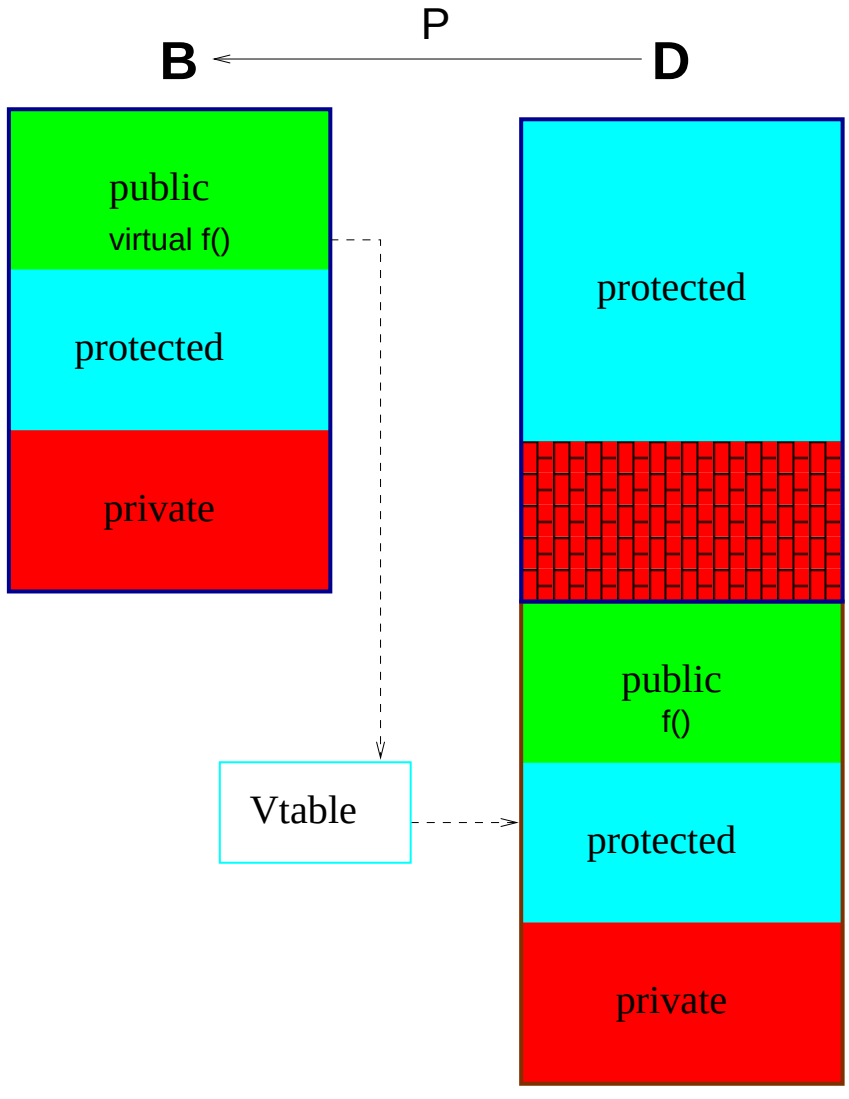
Un modo alternativo è usare la *run time identification* (RTI) attraverso la classe `type_info` della *Standard Library* (libro di testo, paragrafo 8.5.2).

Ereditarietà *protected* e *private*

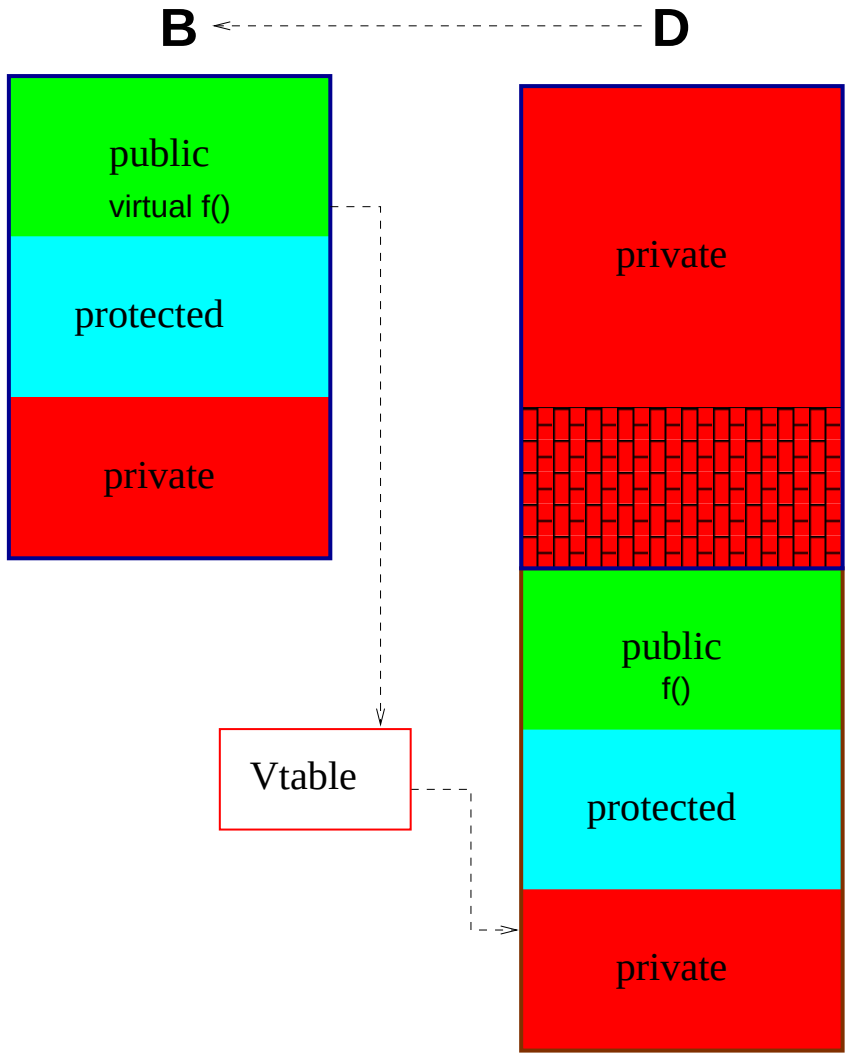
Esistono due altri meccanismi importanti di ereditarietà messi a disposizione dal C++. La differenza dal punto di vista operativo è nel modo in cui viene trattato l'accesso ai membri.

- `class D: protected B`. Membri pubblici e protetti di `B` diventano metodi *protected* di `D`. Solo metodi e *friend* di `D` e di classi derivate possono convertire `D*` in `B*`.
- `class D: private B`. Membri pubblici e protetti di `B` diventano metodi *private* di `D`. Solo metodi e *friend* di `D` possono convertire `D*` in `B*`.

Il meccanismo della ereditarietà protetta



Il meccanismo della ereditarietà privata



Ereditarietà multipla

È possibile ereditare da più di una base, anche con modalità diverse.
Le regole di derivazione si applicano a ciascuna base, così come quelle di costruzione e distruzione.
Se vi è una ambiguità nel nome dei metodi introdotti dalla basi si deve usare il *risolutore di scopo*.

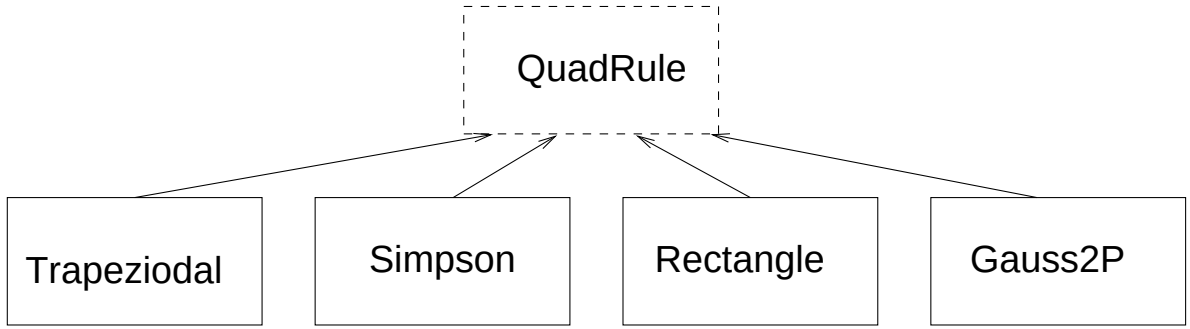
```
class D: public B, protected C{
...
B::pippo(); //uso il metodo di B
```

Aggregazione e delegazione

La ereditarietà pubblica esprime una interazione molto forte tra le classi ed è da utilizzarsi solo quando riflette effettivamente una relazione di tipo *is-a*.

Vi sono altri tipi di interazioni tra i 'concetti' rappresentati dalla classi, che vengono implementati in modo diverso.
Qui parleremo della aggregazione (*has-a*) e delegazione (*is implemented in terms of*), due relazioni di fatto simili.

Supponiamo di avere a disposizione una gerarchia di classi che implementano diverse regole di quadratura, tutte derivate dalla classe astratta `QuadRule` che ne fornisce l'interfaccia pubblica.



Aggregazione

Un elemento finito può essere visto come l'insieme di un elemento finito geometrico, le funzioni di base (e, dualmente, i gradi di libertà), e delle regole di quadratura

```
class FiniteElement
public:
FiniteElement(ElGeo & e, ShapF& f, Quadrule& q)
Elgeo myGeo;
ShapF myShape;
Quadrule myQuad;
;
```

Aggregazione

L'elemento finito è qui visto come un aggregato che mette a disposizione le funzionalità dei tre membri (che sono infatti pubblici).

```
FiniteElement::FiniteElement
(ElGeo & e, ShapF& f, Quadrule& q):
myGeo(e),myShape(f),myQuad(q)
```

Delegazione

Si parla di delegazione quando una classe implementa parte della sua *interfaccia pubblica* attraverso quella di un'altra classe.

Supponiamo di avere una libreria di *matrici* e una libreria di *solutori lineari* e di volere costruire una classe `LinearSystem` che integri le funzionalità di entrambe, e introduca dei metodi addizionali.

```
class LinearSystem{public:
LinearSystem(Matrix & m, LinSolver & l);
Matrix LU() const;//usa un metodo di LinSolver
private:
Matrix * _m;Linsolver * _l;

Matrix LinearSystem::LU() const{
return LinSolver.LU(Matrix);}
```

Modalità di contenimento

Si parla in generale di contenimento (*containement* quando una classe ha tra i suoi membri (tipicamente tra i membri privati o protetti) un oggetto, o un puntatore o una referenza di un'altra classe.

Quali sono i vantaggi/svantaggi delle diverse opzioni?

Contenimento di un oggetto

```
class A{
A(C & c);
protected:
C objectC; };
```

È il caso più semplice. L'oggetto di tipo `C` viene creato nel costruttore di `A`, eventualmente passandolo come parametro. Se `C` contiene nella sua dichiarazione l'espressione `friend class A` allora i metodi di `A` potranno accedere ai membri privati di `C`. Altrimenti `A` ne potrà usare solo l'interfaccia pubblica. Non può essere attivato il polimorfismo su `C`.

Contenimento di un puntatore

```
class A{
A(C * c);
protected:
C * pointerC; };
```

In questo caso, se `C` è una classe base di una gerarchia di classi è possibile usarla in maniera polimorfa (come già visto). Occorre fare attenzione al *lifespan* di `A` e dell'oggetto effettivamente puntato da `pointerC`:

```
C * pc=new C; A(*pc); delete pc
```

è un errore!

N.B.: Questo è un caso dove è indicato l'uso degli smart pointer(per esempio `auto_ptr<>` della Libreria Standard.

Contenimento di un puntatore

Si potrebbe obiettare che l'oggetto puntato da `pointerC` potrebbe essere costruito nel costruttore di `A` e eliminato nel distruttore.
Però in questo modo l'implementazione del polimorfismo *non è più possibile* (in tal caso allora tanto vale memorizzare l'oggetto).

Se si volesse usare la *friendship* per accedere direttamente ai membri privati di `* pointerC` occorre ripetere `friend class A;` nella definizione di TUTTE la classi della gerarchia `C`.
Vale infatti la regola: *l'amico di un mio parente non è necessariamente amico mio*.

Contenimento di una referenza

```
class A{
A(C & c);
protected:
C & refC; };
```

Anche in questo caso `C` è usabile in maniera polimorfa (come già visto) ed occorre fare attenzione al *lifespan* di `A` e dell'oggetto di cui `refC` è una referenza.

Per di più `refC` deve necessariamente essere inizializzato nel costruttore (non esiste la referenza nulla!) e non può essere riassegnata: `A::A(C & c):refC(c){...}`.

Delegazione con ereditarietà privata

La delegazione si può implementare anche con la eredità privata o protetta. In questo caso ci si può avvantaggiare di un uso particolare della istruzione `using`:

```
class D: protected B{
public:
...
using B::fun;
...
```

Il nome `fun` definito nella classe `B` diventa ora parte della interfaccia pubblica di `D`. Supponiamo che in `B` sia stata definita `double fun(double)`. Il codice seguente è corretto:

```
D myD; double x=myD.fun(7.0);
```

Come non spararsi nel piede?

- Usare l'ereditarietà pubblica solo per implementare la relazione IS-A : `D` non deve richiedere di più ne promettere di meno di `B`. In altre parole un utilizzatore di un puntatore alla classe base `B` deve poter utilizzare con sicurezza l'interfaccia pubblica di `B`, anche in caso di puntatori/referenzi polimorfiche.
- Preferire il contenimento semplice per esprimere la relazione di delegazione o di aggregazione. Usare puntatori o referenze solo se la classe contenuta è polimorfa.
- Usare l'ereditarietà protetta o privata solo quando necessaria o chiaramente vantaggiosa (vedi la prossima slide).

Contenimento o ereditarietà protetta/xprivata?

Nel libro di H. Sutter*Exceptional C++* sono contenute delle linee guida che qui si sintetizziamo.

Usare l'ereditarietà privata `B` $\leftarrow$ `D` se

- È necessario sovrascrivere dei metodi virtuali di `B`
- Dobbiamo accedere a un metodo protetto di `B` (alternativa: usare *friend*, ma dobbiamo avere accesso al codice di `B`!)
- La classe `B` contiene solo metodi (empty class optimization)
- Vogliamo un *polimorfismo controllato*, cioè applicabile solo da metodi di `D` e classi derivate da `D` (in questo caso serve la ereditarietà protetta).

Negli altri casi, il contenimento è da preferirsi.