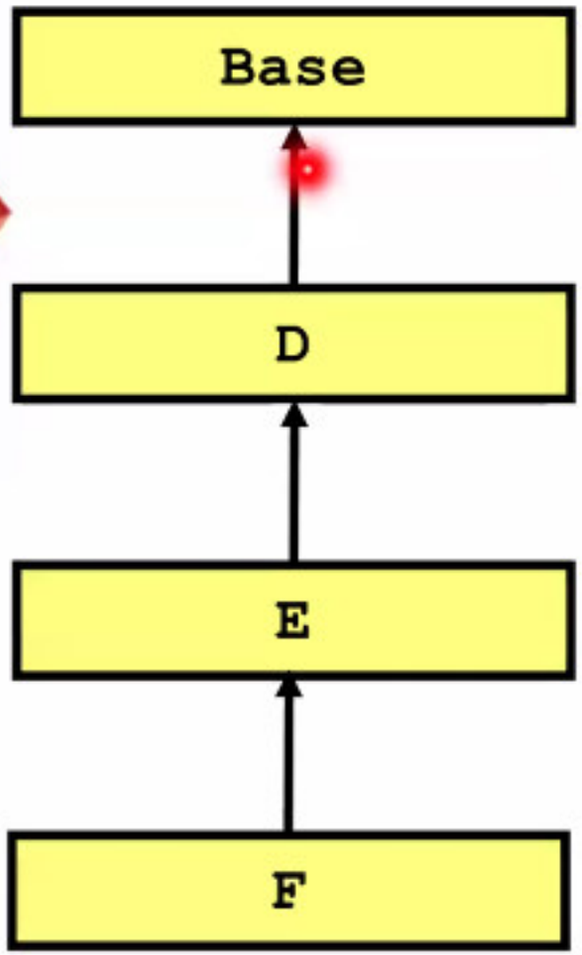
Gerarchie di classi: sottotipi diretti ed indiretti



Casi d'uso di ereditarieta

1) Estensione
2) Specializzazone
3) Ridefinizione
4) Riutilizzo di codice

Ereditarietà per **estensione**

```
dataora <: orario
```

Ereditarietà per **specializzazione**

```
QButton <: QComponent
```

Ereditarietà per **ridefinizione**

```
Queue <: List
```

Ereditarietà per **riuso di codice** **non è subtyping**

`Queue reuse List`

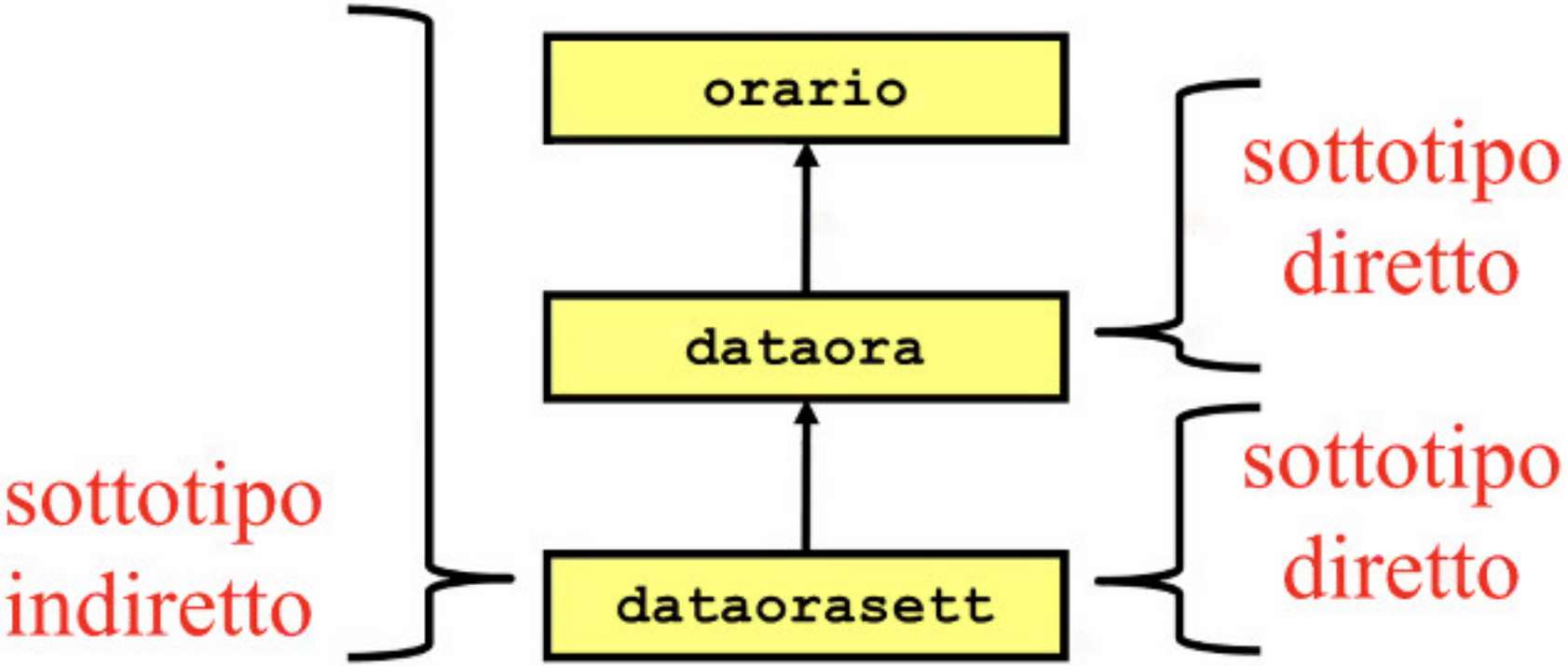
Se vogliamo definire un tipo che oltre alle proprietà di **dataora** memorizzi anche il giorno della settimana possiamo farlo con la seguente derivazione di classe:

```
// tipo enumerazione giorno
enum giorno {lun,mar,mer,gio,ven,sab,dom};

class dataorasett : public dataora {
public:
  giorno GiornoSettimana() const;
private:
  giorno giornosettimana;
};
```

Tipo user-defined **enum**

Una semplice gerarchia di tre classi.



Data una classe **`B`**, per ogni sottotipo **`D`** (in generale indiretto) di **`B`** valgono quindi le seguenti **conversioni implicite**:

`D` $\Rightarrow$ **`B`**	(oggetti)
`D&` $\Rightarrow$ **`B&`**	(riferimenti)
`D*` $\Rightarrow$ **`B*`**	(puntatori)

Grazie alla conversione implicita

dataora ⇒ orario

per valore

possiamo scrivere:

```
int F(orario o) {...}
dataora d;
int i = F(d);
```

Grazie alla conversione implicita

`dataora` $\Rightarrow$ `orario`

possiamo scrivere:

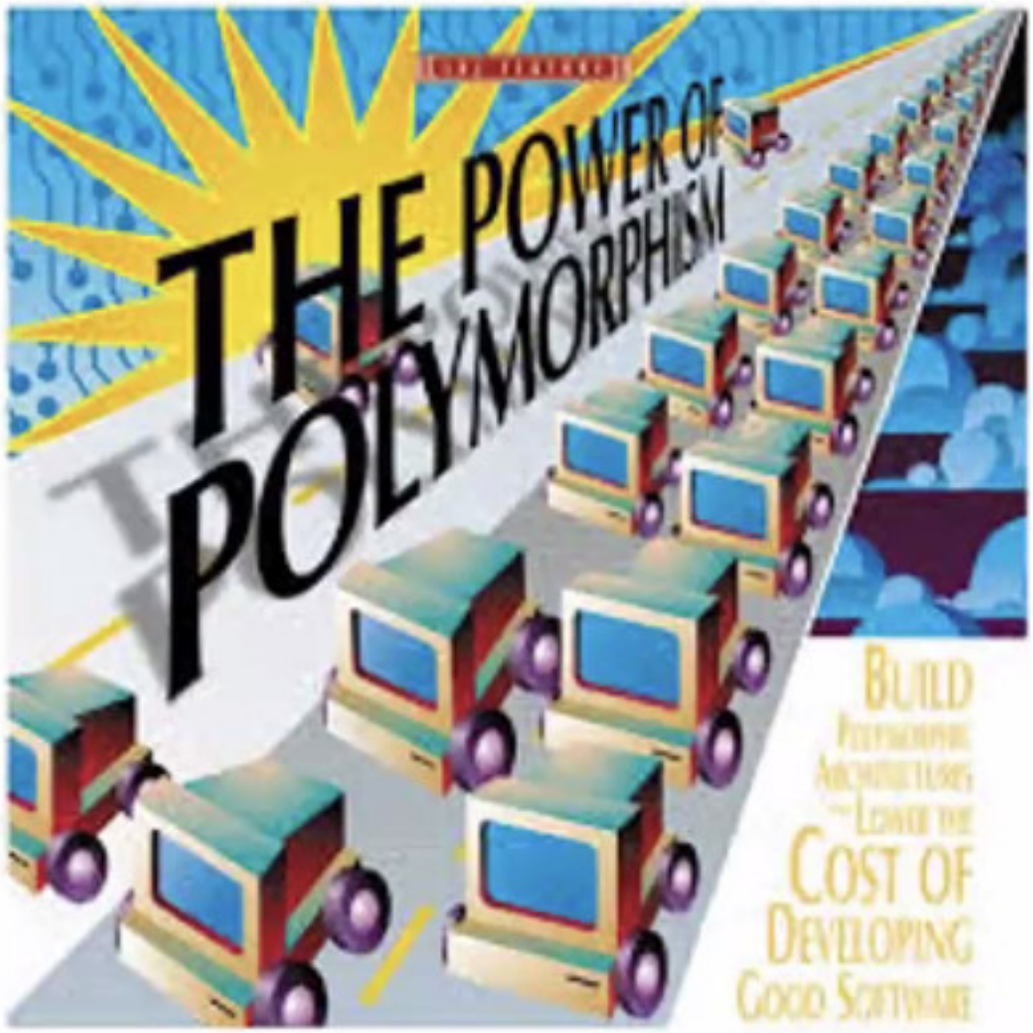
```
int F(orario o) {...}
dataora d;
int i = F(d);
```

Il viceversa non vale!

```
int G(dataora d) {...}
orario o;
int i = G(o); // ILLEGALE
```

Un `dataora` è (in particolare) un `orario`, mentre un `orario` non è un `dataora`!

Polimorfismo in C++ mediante *puntatori* e *riferimenti* e non oggetti

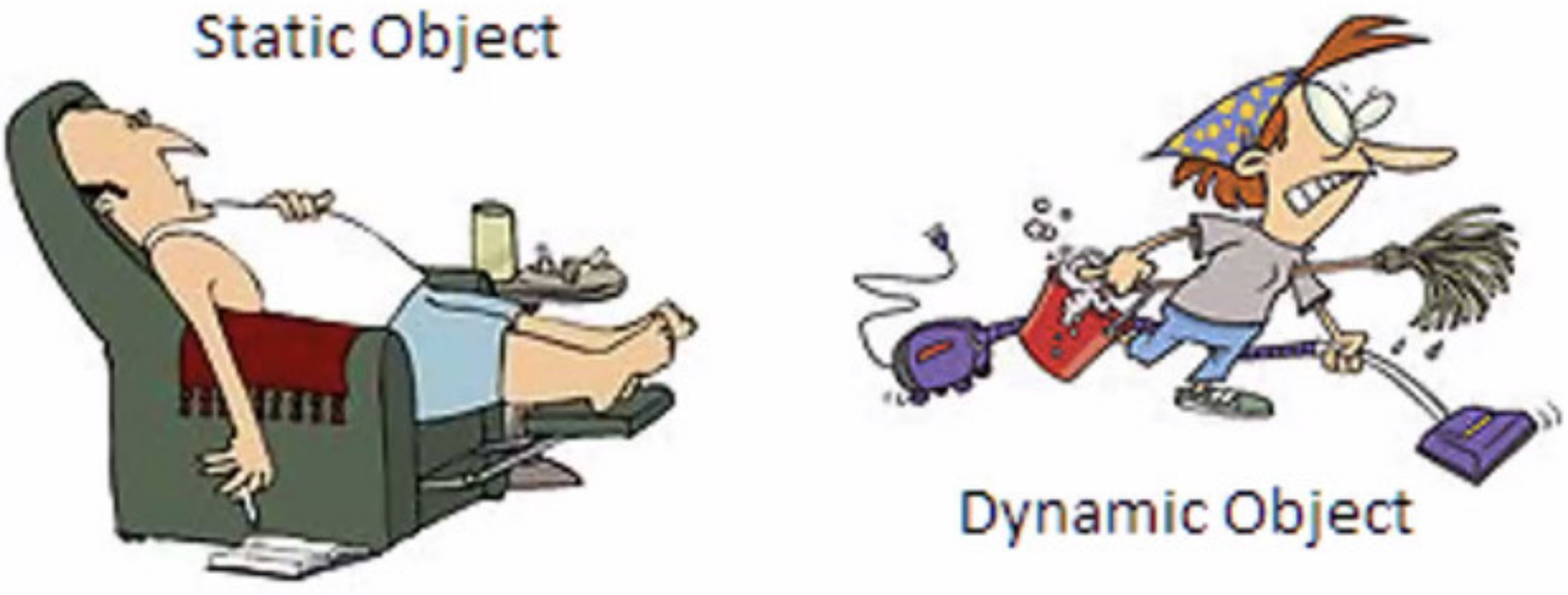


Sia **D** una sottoclasse di **B**.

conversione **D*** $\Rightarrow$ **B***

```
D d; B b;
D* pd=&d;
B* pb=&b;
pb=pd;
```

tipo statico del puntatore **pb** versus *tipo dinamico* di **pb**



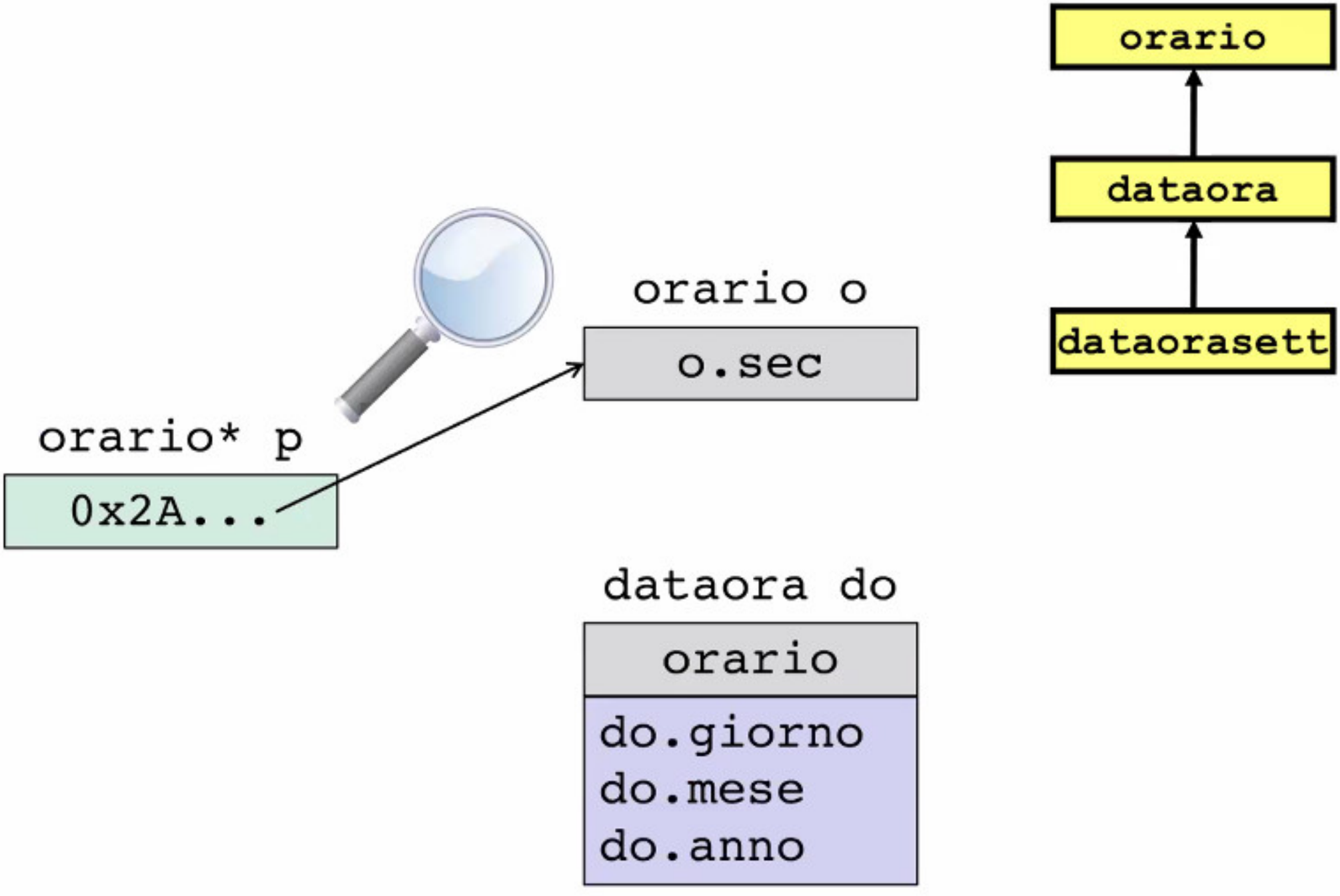
Sia **D** una sottoclasse di **B**.

```
D d; B b;
D* pd=&d;
B* pb=&b;
pb=pd;
```

tipo statico del puntatore **pb** versus *tipo dinamico* di **pb**

Quindi: il **tipo statico** di un puntatore **p** è il tipo **T*** di dichiarazione di **p**, mentre se in un certo istante dell'esecuzione il tipo dell'oggetto a cui effettivamente punta **p** è **U** allora in quell'istante **U*** è il **tipo dinamico** di **p**.

Mentre il tipo statico è fissato al momento della dichiarazione, il tipo dinamico in generale può variare a run-time.

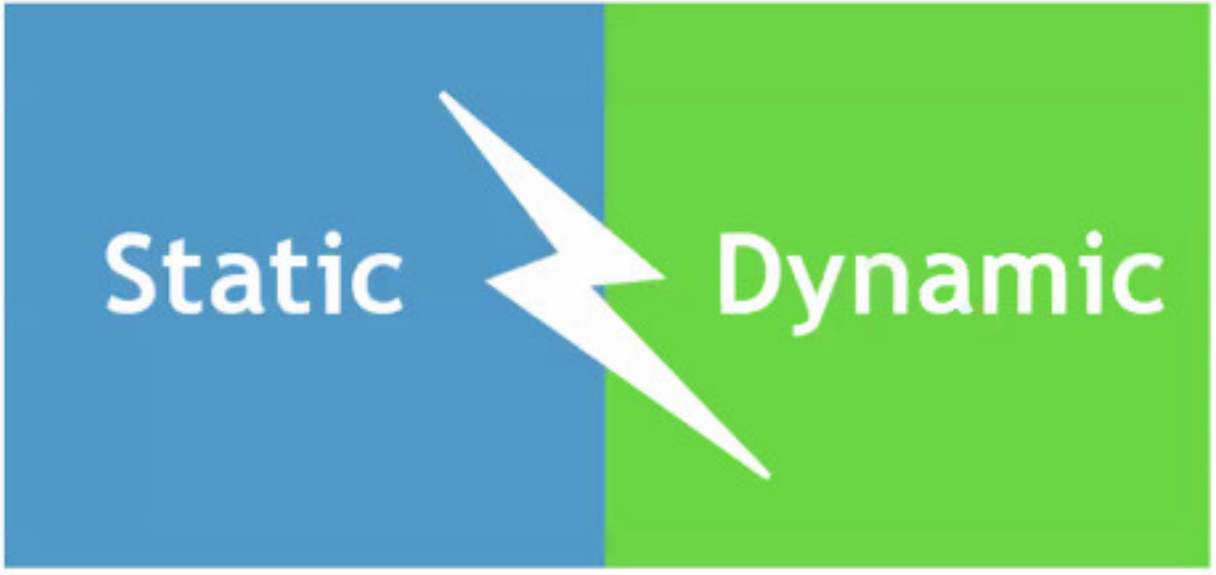

orario
dataora
dataorasett
orario o
o.sec
orario* p
0x2A...
dataora do
orario
do.giorno
do.mese
do.anno

Concetti e terminologia analoghi valgono per i riferimenti:

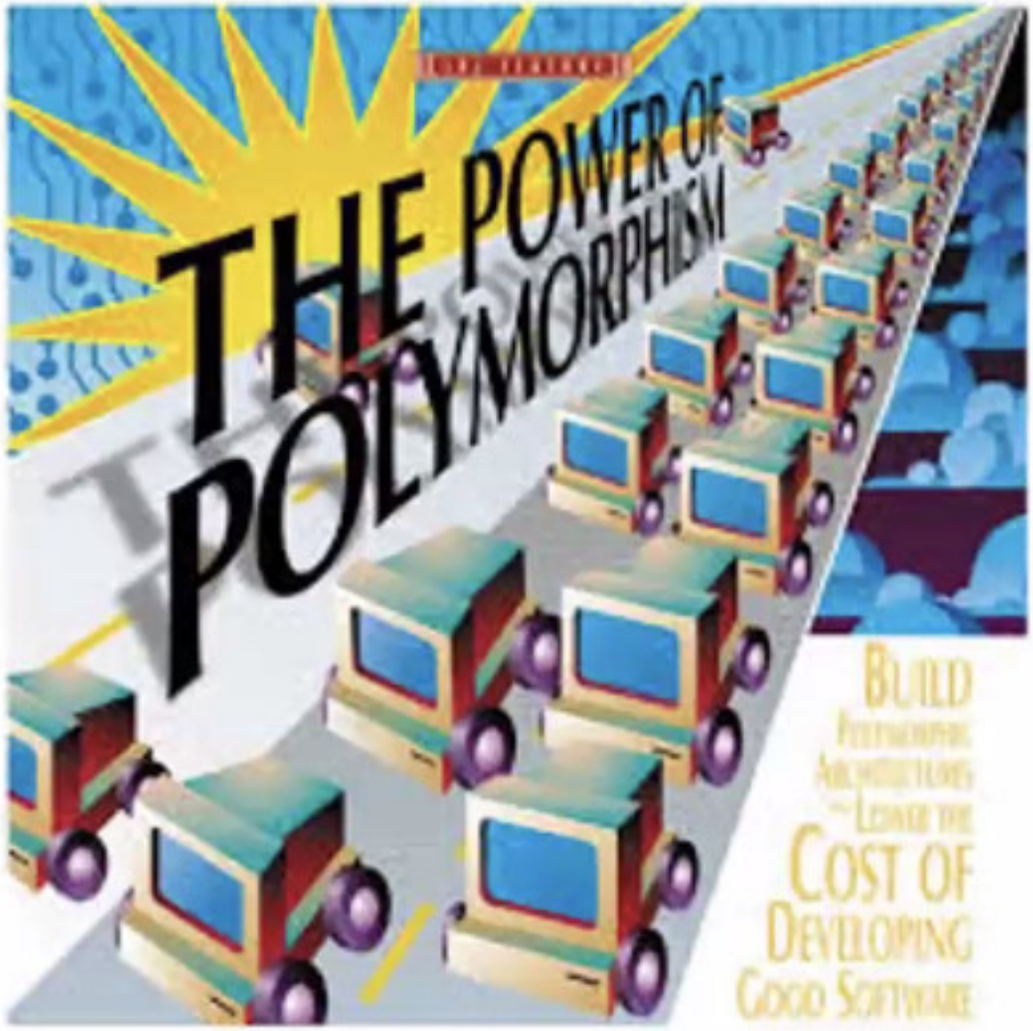
conversione **D& ⇒ B&**

```
D d; B b;
D& rd=d;
B& rb=d;

// D& è il tipo dinamico di rb
```



Studieremo a fondo le benifiche potenzialità del polimorfismo



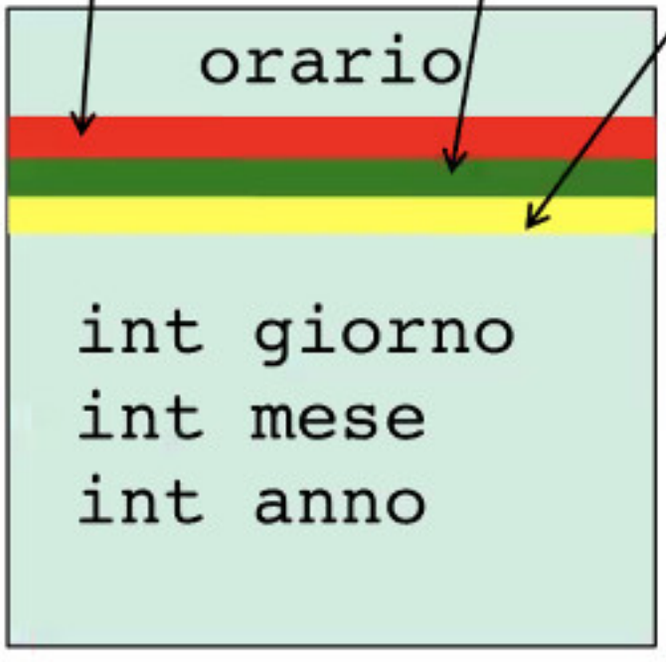
La parte privata della classe base è inaccessibile alla classe derivata

Aggiungendo alla classe **dataora** un metodo **Set2K()** che assegna all'oggetto di invocazione le ore 00:00:00 del 1 gennaio 2000 **non possiamo scrivere**

```
dataora::Set2K() {
   sec = 0; // NO, illegale!
   giorno = 1;
   mese = 1;
   anno = 2000;
}
```

PROTECTED

interfaccia **privata** **pubblica** **protetta**



oggetto `dataorario`

Ereditarietà di tipo

Classe base	*Classe derivata con derivazione*		
membro	**pubblica**		
privato	inaccess.		
protetto	protetto		
pubblico	pubblico		

Ereditarietà di tipo **Ereditarietà di implementazione**

Classe base	*Classe derivata con derivazione*		
membro	**pubblica**		**privata**
privato	~~inaccess.~~		inaccess.
protetto	~~protetto~~		privato
pubblico	~~pubblico~~		privato

Ereditarietà di tipo **Ereditarietà di implementazione**

Classe base	*Classe derivata con derivazione*		
membro	**pubblica**	**protetta**	**privata**
privato	~~inaccess.~~	inaccess.	inaccess.
protetto	~~protetto~~	protetto	privato
pubblico	~~pubblico~~	protetto	privato

Attenzione: Derivazioni protette e private **non supportano** l'ereditarietà di tipo

Attenzione

I membri protected rappresentano comunque una violazione dell'information hiding


```
class C {
private:
  int priv;
protected:
  int prot;
public:
  int publ;
};

class D: private C {
  // prot e publ divengono privati
};

class E: protected C {
  // prot e publ divengono protetti
};

class F: public D {
  // prot e publ sono qui inaccessibili
public:
  void fF(int i, int j){
    // prot=i; // Illegale
    // publ=j; // Illegale
  }
};
class G: public E {
  // prot e publ rimangono qui protetti
  void fG(int i, int j){
    prot=i; // OK
    publ=j; // OK
  }
};
```

Ereditarietà privata

Da un noto libro di Scott Meyers (**Effective C++**):

Ereditarietà privata significa "essere implementati in termini di". Se D deriva privatamente da B significa che in D si è interessati ad alcune funzionalità di B e non si è interessati ad una **relazione concettuale di subtyping** tra D e B. L'ereditarietà privata eredita l'implementazione di B ma non l'interfaccia di B.

L'ereditarietà privata non gioca alcun ruolo nella fase di progettazione del software ma solo nella fase di implementazione del software.

Ereditarietà privata vs relazione has-a

Ereditarietà privata vs relazione has-a

```
class Motore {
private:
  int numCilindri;
public:
  Motore(int nc): numCilindri(nc) {}
  int getCilindri() const {return numCilindri;}
  void accendi() const {
    cout << "Motore a " << getCilindri() << " cilindri acceso" << endl;}
};
```

Relazione has-a

```
class Motore {
private:
  int numCilindri;
public:
  Motore(int nc): numCilindri(nc) {}
  int getCilindri() const {return numCilindri;}
  void accendi() const {
    cout << "Motore a " << getCilindri() << " cilindri acceso" << endl;}
};

Class Auto {
private:
  Motore mot; // Auto has-a Motore come campo dati
public:
  Auto(int nc = 4): mot(4) {}
  void accendi() const {
    mot.accendi();
    cout << "Auto con motore a " << mot.getCilindri() << " cilindri accesa" << endl;
  }
};
```

Ereditarietà privata

```
class Motore {
private:
  int numCilindri;
public:
  Motore(int nc): numCilindri(nc) {}
  int getCilindri() const {return numCilindri;}
  void accendi() const {
    cout << "Motore a " << getCilindri() << " cilindri acceso" << endl;}
};

Class Auto: private Motore { // Auto has-a Motore come sottoogetto
public:
  Auto(int nc = 4): Motore(nc) {}
  void accendi() const {
    Motore::accendi();
    cout << "Auto con motore a " << getCilindri() << " cilindri accesa" << endl;
  }
};
```

Ereditarietà privata vs relazione di composizione has-a

Similarità

1) In entrambi i casi un oggetto Motore "contenuto" in ogni oggetto Auto

2) In entrambi i casi, per gli utenti esterni, Auto* non è convertibile a Motore*

Differenze

1) La composizione è necessaria se servono **più motori** in un auto (a meno di usi limite di ereditarietà multipla)

2) Ered.privata può introdurre **ereditarietà multipla** (problematica) **non necessaria**

3) Ered.privata **permette ad Auto** di convertire Auto* a Motore*

4) Ered.privata permette **l'accesso alla parte protetta** della base

Conversioni implicite e tipologia di derivazione

Le conversioni implicite indotte dalla derivazione **valgono solamente per la derivazione pubblica** che è quindi l'unica tipologia di derivazione che supporta la relazione “is-a”. La derivazione protetta e la derivazione privata **non inducono** alcuna conversione implicita.

Conversioni implicite e tipologia di derivazione

Le conversioni implicite indotte dalla derivazione **valgono solamente per la derivazione pubblica** che è quindi l'unica tipologia di derivazione che supporta la relazione "is-a". La derivazione protetta e la derivazione privata **non inducono** alcuna conversione implicita.

```
class C {
private:
  int i;
protected:
  char c;
public:
  float f;
};

class D: private C { };    // derivazione privata
class E: protected C { }; // derivazione pubblica
// nessuna conversione implicita D => C e E => C
int main() {
  C c, *pc; D d, *pd; E e, *pe;
//c=d;      // Illegale: "C is an inaccessible base of D"
//c=e;      // Illegale
//pc=&d;    // Illegale
//pc=&e;    // Illegale
//C& rc=d; // Illegale
}
```

Ereditare i metodi pubblici

```
class Base {
  int x;
public:
  void f() {x=2;}
};

class Derivata: public Base {
  int y;
public:
  void g() {y=3;}
};

int main() {
  Base b; Derivata d;
  Base* p = &b; Derivata* q = &d;
  p->f(); // OK
  p=&d;   // Derivata* è ora il tipo dinamico di p
  p->f(); // OK
  p->g(); // HA SENSO?
  q->g(); // OK
  q->f(); // OK
}
```

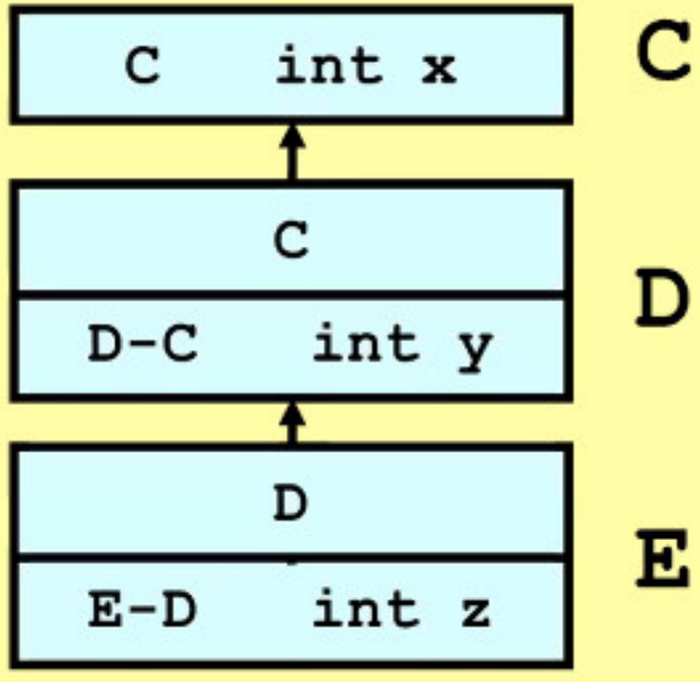
Conversioni Base*⇒Derivata*, Base&⇒Derivata&

```
// VI significa Valore (intero) Imprevedibile
class C {
public:
  int x;
  void f() {x=4;}
};

class D: public C {
public:
  int y;
  void g() {x=5; y=6;}
};

class E: public D {
public:
  int z;
  void h() {x=7; y=8; z=9;}
};

int main() {
  C c; D d; E e;
  c.f(); d.g(); e.h();
  D* pd = static_cast<D*> (&c); // PERICOLOSO!
  cout << pd->x <<" "<< pd->y << endl; // errore run-time o stampa: 4 VI
  E& re = static_cast<E&> (d);  // PERICOLOSO!
  cout << re.x <<" "<< re.y <<" "<< re.z << endl; // err. run-time o stampa: 5 6 VI
  C* pc = &d; pd = static_cast<D*> (pc); // OK
  cout << pd->x <<" "<< pd->y << endl; // stampa: 5 6
  D& rd = e; E& re1 = static_cast<E&> (rd); // OK
  cout << re1.x <<" "<< re1.y <<" "<< re1.z << endl; // stampa: 7 8 9
}
```



Google falsa amicizia

Tutti Immagini

Ereditarietà e amicizia: le amicizie non si ereditano!

Ereditarietà e amicizia: le amicizie non si ereditano!

```
class C {
private:
  int i;
public:
  C(): i(1) {}
  friend void print(C);
};

class D: public C {
private:
  double z;
public:
  D(): z(3.14) {}
};

void print(C x) {
  cout << x.i << endl;
  D d;
//cout << d.z << endl; // Illegale:
//"D::z is private within this context"
}

int main() {
  C c; D d;
  print(c); // stampa: 1
  print(d); // OK, stampa: 1
}
```

Ereditarietà e amicizia: le amicizie non si ereditano!

```
class C {
  friend class Z;
private:
  int i;
public:
  C(): i(1) {}
};

class D: public C {
private:
  double z;
public:
  D(): z(3.14) {}
};

class Z {
public:
  void m() { C c; D d; cout << c.i; // OK
  cout << d.z; // Illegale: "D::z is private within this context"
  }
};

int main() {
  Z z;
  z.m(); // stampa: 1
}
```

Ereditarietà e amicizia

```
class C {
private:
  int i;
public:
  C(): i(1) {}
  friend void print(C);
};
void print(C x) {cout << x.i << endl;}

class D: public C {
private:
  double z;
public:
  D(): z(3.14) {}
  friend void print(D);
};
void print(D x) {cout << x.z << endl;}

int main() {
  C c; D d;
  print(c); // stampa: 1
  print(d); // stampa: 3.14
}
```

Sul significato di inaccessibile

Sul significato di inaccessibile

```
class C {
private:
  int i;
public:
  C(): i(1) {}
  void print() {cout << ' ' << i;}
};

class D: public C {
private:
  double z;
public:
  D(): z(3.14) {}
  void print() {
    C::print(); // l'oggetto di invocazione di C::print() è il
                // sottooggetto di tipo C dell'oggetto di invocazione
  //cout << ' ' << this->i; // membro i INACCESSIBILE
    cout << ' ' << z;}
};

int main() {
  C c; D d;
  c.print(); cout << endl;    // stampa: 1
  d.print(); cout << endl;    // stampa: 1 3.14
}
```