

## Sul significato di protected



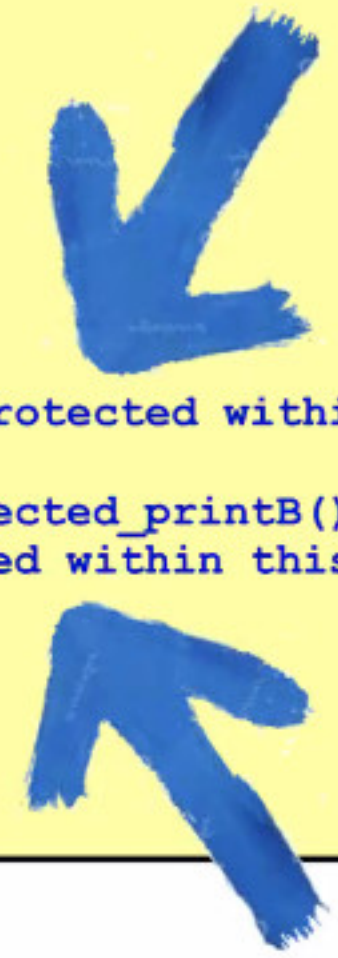
## Sul significato di protected

```
class B {
protected:
    int i;
    void protected_printB() const {cout << ' ' << i;}
public:
    void printB() const {cout << ' ' << i;}
};

class D: public B {
private:
    double z;
public:
    void stampa() {
        cout << i << ' ' << z; // OK
    }

    static void stampa(const B& b,const D& d) {
        cout << ' ' << b.i; // Illegale: "B::i is protected within this context"
        b.printB(); // OK
        b.protected_printB(); // Illegale: "B::protected_printB() is
                                // protected within this context"

        cout << ' ' << d.i; // OK
        d.printB(); // OK
        d.protected_printB(); // OK
    }
};
```



# Ridefinizione di metodi

Potrebbe avere senso ridefinire nella classe derivata alcune funzionalità ereditate dalla classe base.

I metodi sono concepiti come dei contratti, quindi l'implementazione di un contratto della classe base potrebbe richiedere **variazioni o adattamenti** nella classe derivata





## Ridefinizione di `orario::operator+`



```
dataora dataora::operator+(const orario& o) const {  
    dataora aux = *this;  
    // ATTENZIONE:  
    // aux.sec = sec + o.sec; darebbe un errore di compilazione!  
    // perchè anche se sec è dichiarato protected in orario,  
    // o.sec è comunque inaccessibile in dataora  
    aux.sec = sec + 3600*o.Ore() + 60*o.Minuti() + o.Secondi();  
    if (aux.sec >= 86400) {  
        aux.sec = aux.sec - 86400;  
        aux.AvanzaUnGiorno();  
    }  
    return aux;  
}
```

```
void dataora::AvanzaUnGiorno() { // metodo proprio  
    if (giorno < GiorniDelMese()) giorno++;  
    else if (mese < 12) { giorno = 1; mese++; }  
    else { giorno = 1; mese = 1; anno++; }  
}
```

Possiamo invocare l'operatore + di **orario** o **dataora** nel modo seguente:

```
orario o1, o2;  
dataora d1, d2;  
o1 + o2;      // invoca orario::operator+  
d1 + d2;      // invoca dataora::operator+  
o1 + d2;      // invoca orario::operator+  
d1 + o2;      // invoca dataora::operator+  
orario y = d1 + d2; // OK  
dataora x = o1 + o2; // Illegale  
d1.orario::operator+(d2); // invoca orario::operator+
```



## *Name hiding rule*

Una ridefinizione in  $D$  del nome di metodo  $m()$  nasconde sempre tutte le versioni sovraccaricate di  $m()$  disponibili in  $B$ , che non sono quindi direttamente accessibili in  $D$  ma solamente tramite l'operatore di scoping  $B::$



Se ridefiniamo il metodo `ore` in `dataora` con segnatura:

```
int dataora::Ore(int) const {  
    ...  
}
```

non possiamo più scrivere:

```
dataora d;  
cout << d.Ore(); // Illegale
```

perchè il “vecchio” metodo `Ore` della classe `orario` è **mascherato** in `dataora` dalla ridefinizione. Per l’accesso possiamo però usare l’operatore di scoping:

```
dataora d;  
cout << d.orario::Ore();
```

**Ridefinizione di campi dati**

**R**edefinition

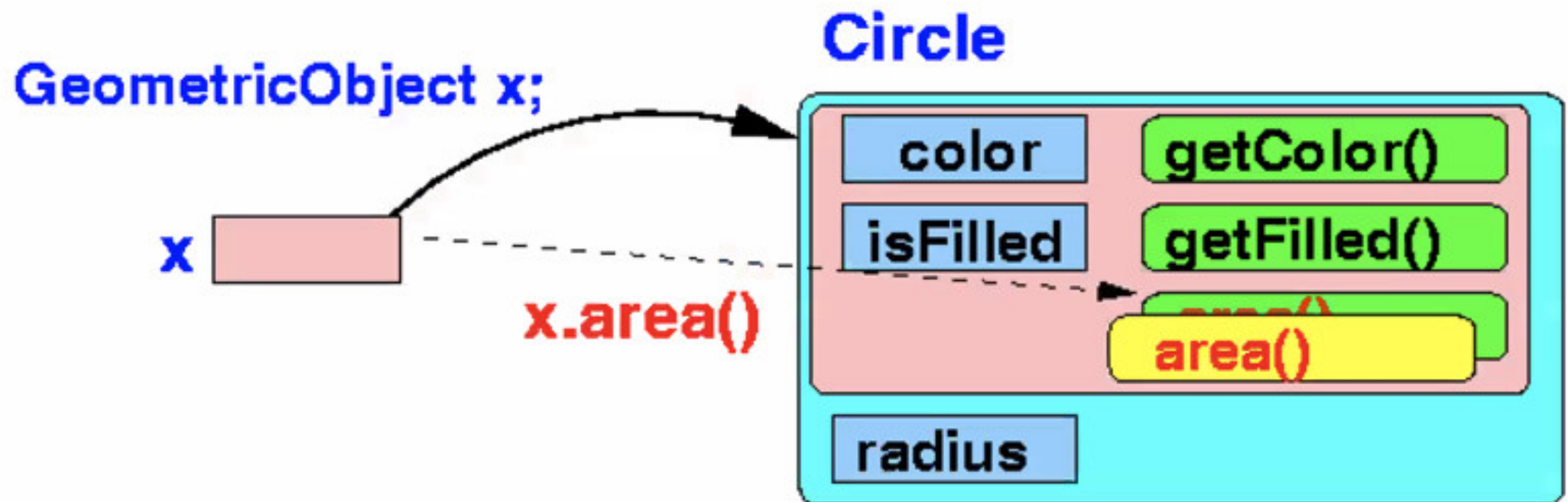


```
class B {
protected:
    int x;
public:
    B() : x(2) {}
    void print() {cout << x << endl;}
};

class D: public B {
private:
    double x; // ridefinizione del campo dati x
public:
    D() : x(3.14) {}
    // ridefinizione di print()
    void print() {cout << x << endl;} // è D::x
    void printAll() {cout << B::x << ' ' << x << endl;}
};

main () {
    B b; D d;
    b.print(); // stampa: 2
    d.print(); // stampa: 3.14
    d.printAll(); // stampa: 2 3.14
}
```

# *Static binding* nell'invocazione di metodi



*Static binding*

```
class Base {
    int x;
public:
    void f() {x=2;}
};
class Derivata: public Base {
    int y;
public:
    void f() { Base::f(); y=3; } // ridefinizione
};
int main() {
    Base b; Derivata d;
    Base* p = &b;
    p->f(); // invoca Base::f()
    p=&d;   // Derivata* è il tipo dinamico di p
    p->f(); // cosa invoca??
}
```



**Base::f()**



```

class B {
public:
    int f() const { cout << "B::f()\n"; return 1; }
    int f(string) const { cout << "B::f(string)\n"; return 2; }
};

class D : public B {
public:
    // ridefinizione con la stessa segnatura
    int f() const { cout << "D::f()\n"; return 3; }
};

class E : public B {
public:
    // ridefinizione con cambio del tipo di ritorno
    void f() const { cout << "E::f()\n"; }
};

class H : public B {
public:
    // ridefinizione con cambio lista argomenti
    int f(int) const { cout << "H::f()\n"; return 4; }
};

int main() {
    string s; B b; D d; E e; H h;
    int x = d.f(); // stampa: D::f()
    //d.f(s);      // Illegale
    //x = e.f();    // Illegale
    //x = h.f();    // Illegale
    x = h.f(1);    // stampa: H::f()
}

```



```
class C {  
public:  
    void f(int x) { }  
    void f() { }  
};  
class D: public C {  
    int y;  
public:  
    void f(int x) {f(); y=3+x;}  
    // Illegale:  
    // "no matching function for D::f()"  
};
```

```

class C {
public:
    int x;
    void f() {x=1;}
};

class D: public C {
public:
    int y;
    void f() {C::f(); y=2;} // OK
};

int main() {
    C c; D d; c.f(); d.f();
    cout << c.x << endl; // stampa: 1
    cout << d.x << " " << d.y;
    // stampa: 1 2
}

```

```

class C {
public:
    int x;
    void f() {x=1;}
};

class D: public C {
public:
    int y;
    void f() {std::cout <<"*";
              y=3; f();}
              // errore logico:
              // ricorsione infinita!
};

int main() {
    D d; d.f(); // compila ma...
} // errore run-time:
    // è uno stack overflow!

```

Domanda: " se io ho più livelli di ereditarietà (B, D1, D2) e voglio usare un metodo f() di B, ridefinito sia in D1 che in D2, da D2 la chiamata sarà d2.B::f() ? "

```
class B {  
public:  
void f() {}  
};  
  
class D1: public B {  
public:  
void f() {}  
};  
  
class D2: public D1 {  
public:  
void g() {  
D1::f(); // D1::f()  
B::f(); // B::f()  
}  
  
void f() {  
f(); // chiamata ricorsiva D2::f()  
D1::f();  
B::f();  
}  
};
```

```

class C {
public:
    void f() {cout << "C::f\n";}
};

class D: public C {
public:
    void f() {cout << "D::f\n";}; // ridefinizione
};

class E: public D {
public:
    void f() {cout << "E::f\n";}; // ridefinizione
};

int main() {
    C c; D d; E e;
    C* pc = &c; E* pe = &e;
    c = d;      // OK: conversione D => C
    c = e;      // OK: conversione E => C
    d = e;      // OK: conversione E => D
    C& rc=d;    // OK: conversione D => C
    D& rd=e;    // OK: conversione E => D
    pc->f();    // OK
    pc = pe;    // OK: conversione E* => C*
    rd.f();     // OK
    c.f();      // OK
    pc->f();    // OK
}

```



# Costruttori, distruttori, assegnazioni

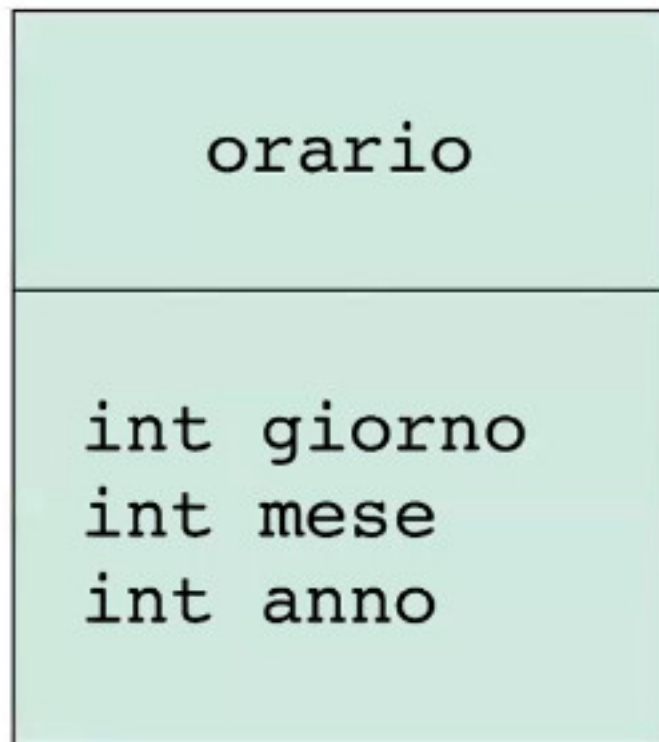


Value = 20;



Assign 20  
to  
Variable value

## Costruttori, distruttori, assegnazioni nelle classi derivate



oggetto dataora

## Costruttori nelle classi derivate

La lista di inizializzazione di un costruttore di una classe D derivata direttamente da B in generale può contenere invocazioni di costruttori per i campi dati (propri) di D e l'invocazione di un costruttore della classe base B.

L'esecuzione di un tale costruttore di D avviene nel seguente modo:

**[1]** viene sempre e comunque invocato per primo un costruttore della classe base B, o esplicitamente o implicitamente il costruttore di default di B quando la lista di inizializzazione non include una invocazione esplicita;

**[2]** successivamente, secondo il comportamento già noto, viene eseguito il costruttore “proprio” di D, ossia vengono costruiti i campi dati propri di D;

**[3]** infine viene eseguito il corpo del costruttore.



In particolare, se nella classe derivata D si omette qualsiasi costruttore allora, come al solito, è disponibile il **costruttore di default standard** di D. Il suo comportamento è quindi il seguente:

- [1]** richiama il costruttore di default di B;
- [2]** successivamente si comporta come il costruttore di default standard “proprio” di D, ossia richiama i costruttori di default per tutti i campi dati di D



```
dataora::dataora(): giorno(1), mese(1), anno(2000) {}
```

```
dataora d;  
cout << d.Ore(); // stampa: 0  
cout << d.Giorno(); // stampa: 1
```

È naturale definire il seguente costruttore con parametri per la classe derivata `dataora`.

```
dataora::dataora(int a, int me, int g, int o, int m, int s)  
    : orario(o,m,s), giorno(g), mese(me), anno(a) {}
```

```
dataora d(2020,11,17,11,55,13);  
cout << d.Ore(); // stampa: 11  
cout << d.Giorno(); // stampa: 17
```



```
class Z {  
public:  
    Z() {cout << "Z0 ";}  
};  
  
class C {  
private:  
    int x;  
public:  
    C(int z=1): x(z) {cout << "C01 ";}  
};  
  
class D: public C {  
private:  
    int y;  
    Z z;  
};  
  
int main() {  
    D d; // costruttore standard  
}  
// stampa: C01 Z0
```



```
class Z {
public:
    Z() {cout << "Z0 ";}
    Z(double d) {cout << "Z1 ";}
};

class C {
private:
    int x;
    Z w;
public:
    C(): w(6.28), x(8) {cout << x << " C0 ";}
    C(int z): x(z) {cout << x << " C1 ";}
};

class D: public C {
private:
    int y;
    Z z;
public:
    D(): y(0) {cout << "D0 ";}
    D(int a): y(a), z(3.14), C(a) {cout << "D1 ";}
};

int main() {
    D d; cout << "UNO\n";
    D e(4); cout << "DUE";
}

// stampa:
// Z1 8 C0 Z0 D0 UNO
// Z0 4 C1 Z1 D1 DUE
```

## **Costruttore di copia standard nelle classi derivate**

