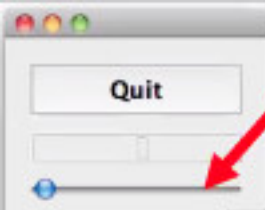


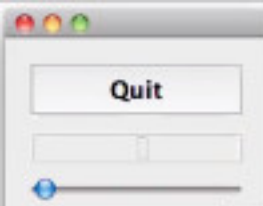
Layout

```
class MyWidget : public QWidget {
public:
    MyWidget(QWidget *parent = 0) : QWidget(parent) {
        QPushButton *quit = new QPushButton(tr("Quit"));
        quit->setFont(QFont("Times", 18, QFont::Bold));
        // QLCDNumber è un widget che mostra numeri come un LCD
        QLCDNumber *lcd = new QLCDNumber(2); // 2 cifre
        // rende lcd più leggibile
        lcd->setSegmentStyle(QLCDNumber::Filled);
    }
};
```



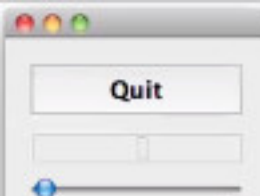
Layout

```
class MyWidget : public QWidget {
public:
    MyWidget(QWidget *parent = 0) : QWidget(parent) {
        QPushButton *quit = new QPushButton(tr("Quit"));
        quit->setFont(QFont("Times", 18, QFont::Bold));
        // QLCDNumber è un widget che mostra numeri come un LCD
        QLCDNumber *lcd = new QLCDNumber(2); // 2 cifre
        // rende lcd più leggibile
        lcd->setSegmentStyle(QLCDNumber::Filled);
        // Il widget QSlider permette di selezionare un valore intero in un range
        // slider è un QSlider orizzontale con valori in [0,99], e valore iniziale 0
        QSlider *slider = new QSlider(Qt::Horizontal);
        slider->setRange(0, 99); slider->setValue(0);
        connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
    }
};
```



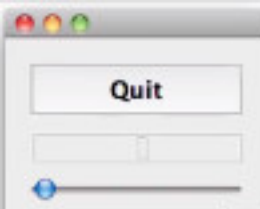
Layout

```
class MyWidget : public QWidget {
public:
MyWidget(QWidget *parent = 0) : QWidget(parent) {
    QPushButton *quit = new QPushButton(tr("Quit"));
    quit->setFont(QFont("Times", 18, QFont::Bold));
    // QLCDNumber è un widget che mostra numeri come un LCD
    QLCDNumber *lcd = new QLCDNumber(2); // 2 cifre
    // rende lcd più leggibile
    lcd->setSegmentStyle(QLCDNumber::Filled);
    // Il widget QSlider permette di selezionare un valore intero in un range
    // slider è un QSlider orizzontale con valori in [0,99], e valore iniziale 0
    QSlider *slider = new QSlider(Qt::Horizontal);
    slider->setRange(0, 99); slider->setValue(0);
    connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
    // segnale valueChanged(int) di slider connesso allo slot display(int) di lcd
    // slider emette il segnale valueChanged() quando cambia il suo valore
    // lo slot è quindi chiamato quando viene emesso questo segnale
    connect(slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)));
}
```



Layout

```
class MyWidget : public QWidget {
public:
MyWidget(QWidget *parent = 0) : QWidget(parent) {
    QPushButton *quit = new QPushButton(tr("Quit"));
    quit->setFont(QFont("Times", 18, QFont::Bold));
    // QLCDNumber è un widget che mostra numeri come un LCD
    QLCDNumber *lcd = new QLCDNumber(2); // 2 cifre
    // rende lcd più leggibile
    lcd->setSegmentStyle(QLCDNumber::Filled);
    // Il widget QSlider permette di selezionare un valore intero in un range
    // slider è un QSlider orizzontale con valori in [0,99], e valore iniziale 0
    QSlider *slider = new QSlider(Qt::Horizontal);
    slider->setRange(0, 99); slider->setValue(0);
    connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
    // segnale valueChanged(int) di slider connesso allo slot display(int) di lcd
    // slider emette il segnale valueChanged() quando cambia il suo valore
    // lo slot è quindi chiamato quando viene emesso questo segnale
    connect(slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)));
    // QVBoxLayout è un layout manager (LM) che gestisce la geometria di piazzamento
    // dei figli di un widget; inoltre il LM di un widget w gestisce il
    // ridimensionamento dei figli di w quando viene ridimensionato w
    // QLayout non è sottotipo di QWidget, quindi un LM non ha mai un parent
    QVBoxLayout *layout = new QVBoxLayout;
    // layout->addWidget() aggiunge dei widget al LM layout
    layout->addWidget(quit); layout->addWidget(lcd); layout->addWidget(slider);
    // QWidget::setLayout(layout) installa layout come LM di this
    // inoltre rende layout un figlio di this, e tutti i widget in layout
    // diventano figli di this
    setLayout(layout);
}
};
```



Layout

```
class MyWidget : public QWidget {
public:
    MyWidget(QWidget *parent = 0) : QWidget(parent) {
        QPushButton *quit = new QPushButton(tr("Quit"));
        quit->setFont(QFont("Times", 18, QFont::Bold));
        // QLCDN
        QLCDNum
        // rende
        lcd->set
        // Il wi
        // slide
        QSlider
        slider->
        connect
        // segna
        // slide
        // lo sl
        connect(slider,
        // QVBoxLayout è
        // dei figli di
        // ridimensionam
        // QLayout non è
        QVBoxLayout *lay
        // layout->addWi
        layout->addWidget
        // QWidget::setL
        // inoltre rende
        // diventano figli di this
        setLayout(layout);
    }
};
```



in LCD

colore intero in un range

-21/pogg/progetto/Qt-learn/lec05

Date Modified	Size	Kind
9 Dec 2020 at 11:01	5 KB	Docu
9 Dec 2020 at 11:01	31 KB	Applic
10 Mar 2020 at 11:37	925 bytes	Qt Pro
6 Jul 2014 at 18:45	923 bytes	C++ s
9 Dec 2020 at 11:01	9 KB	object
9 Dec 2020 at 11:01	55 KB	TextE

0

lcd

zamento

);

Slots

- Gli slots sono metodi “ordinari”, preceduti dalla Qt keyword **slots**
- Una classe contenente propri slots deve includere nella parte privata la macro **QObject::Q_OBJECT**. Serve al MOC
- In ogni file .cpp/.h che contiene la macro **Q_OBJECT**, tutti gli slots saranno espansi dal MOC in un file .moc
- Tutto ciò è automaticamente gestito da qmake

Signals

- Aggiungere sempre alla parte privata di una classe con segnali propri la macro `QObject::Q_OBJECT`. Serve al MOC
- Un segnale si **dichiara** (non si definisce) come un metodo ordinario preceduto dalla Qt keyword `signals`
`signals:`
`void valueChanged(int newValue);`
- Il segnale **non va implementato**
- Per emettere il segnale si usa la Qt keyword `emit`:
`emit valueChanged(12);`

More on Signals and Slots

Tutorato by
Benedetto Cosentino & Alberto Sinigaglia

Construction/Destruction Order of QObjects

When `QObjects` are created on the heap (i.e., created with `new`), a tree can be constructed from them in any order, and later, the objects in the tree can be destroyed in any order. When any `QObject` in the tree is deleted, if the object has a parent, the destructor automatically removes the object from its parent. If the object has children, the destructor automatically deletes each child. No `QObject` is deleted twice, regardless of the order of destruction.

When `QObjects` are created on the stack, the same behavior applies. Normally, the order of destruction still doesn't present a problem. Consider the following snippet:

```
int main()
{
    QWidget window;
    QPushButton quit("Quit", &window);
    ...
}
```

The parent, `window`, and the child, `quit`, are both `QObjects` because `QPushButton` inherits `QWidget`, and `QWidget` inherits `QObject`. This code is correct: the destructor of `quit` is *not* called twice because the C++ language standard (ISO/IEC 14882:2003) specifies that destructors of local objects are called in the reverse order of their constructors. Therefore, the destructor of the child, `quit`, is called first, and it removes itself from its parent, `window`, before the destructor of `window` is called.

Ereditarietà **multipla**



Multiple inheritance

From Wikipedia, the free encyclopedia



This article **needs additional citations for verification**. Please help [improve this article](#) by [adding citations to reliable sources](#). Unsourced material may be challenged and removed.

(August 2012)

Multiple inheritance is a feature of some [object-oriented](#) computer [programming languages](#) in which an object or [class](#) can [inherit](#) characteristics and features from more than one parent object or [parent class](#). It is distinct from single inheritance, where an object or class may only inherit from one particular object or class.

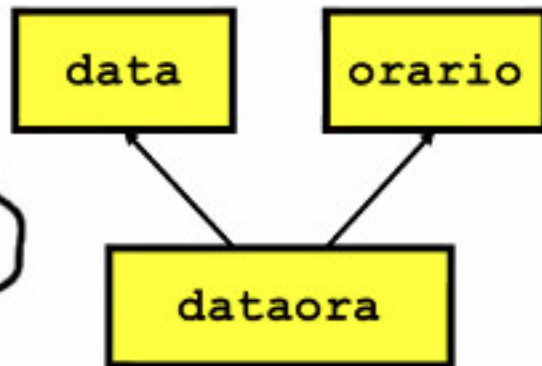
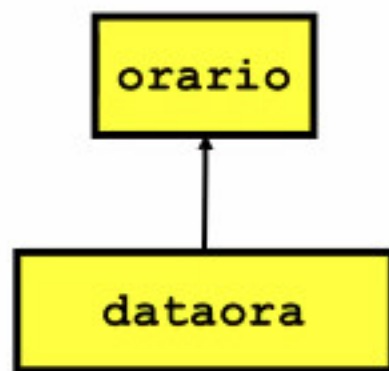
Multiple inheritance has been a [touchy issue](#) for many years^{[\[citation needed\]](#)}, with opponents pointing to its increased [complexity](#) and [ambiguity](#) in situations such as the "[diamond problem](#)", where it may be ambiguous as to which parent class a particular feature is inherited from if more than one parent class implements said feature. This can be addressed in various ways, including using [virtual inheritance](#).^{[\[1\]](#)} Alternate methods of object composition not based on inheritance such as [mixins](#) and [traits](#) have also been proposed to address the ambiguity.

Contents [\[hide\]](#)

- 1 Details
- 2 Implementations
- 3 The diamond problem
 - 3.1 Mitigation
- 4 See also
- 5 References
- 6 Further reading
- 7 External links

Python, Ruby, Scala

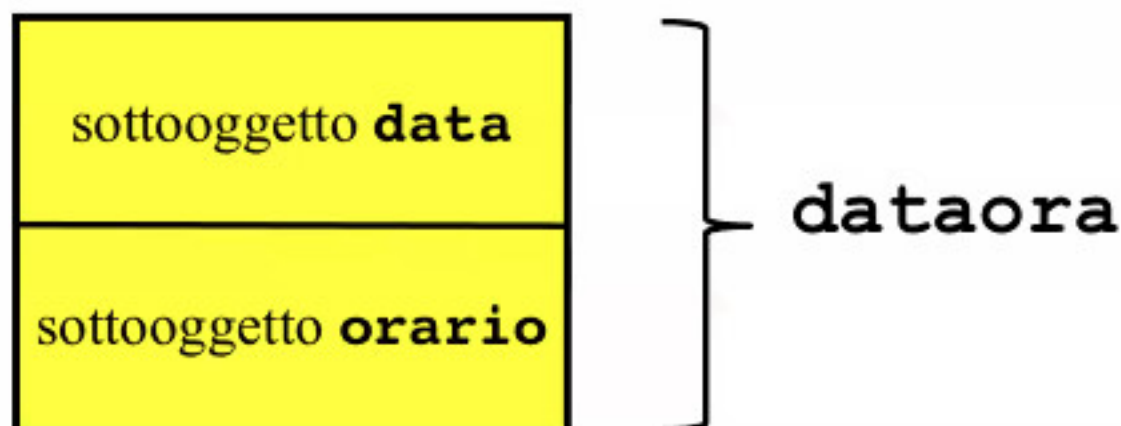
PHP, Rust





```
class data {  
private: // metodi di utilità  
    int GiorniDelMese() const;  
    bool Bisestile() const;  
protected:  
    int giorno, mese, anno;  
    void AvanzaUnGiorno(); // utilità per la gerarchia  
public:  
    data(int =1, int =1, int =0);  
    int Giorno() const { return giorno; }  
    int Mese() const { return mese; }  
    int Anno() const { return anno; }  
};
```


Un oggetto di **dataora** ha due sottooggetti.





```
#include "orario.h"
#include "data.h"

class dataora : public data, public orario {
public:
    dataora() {}
    dataora(int a, int me, int g, int o, int m, int s)
        : data(a, me, g), orario(o, m, s) {}
    dataora operator+(const orario&) const;
    bool operator==(const dataora&) const;
    ...
};
```

Supponiamo di aver sia nella classe `orario` che nella classe `data` un metodo `Stampa()` :

```
void orario::Stampa() const {  
    cout << Ore() << ':' << Minuti() << ':' << Secondi();  
}
```

```
void data::Stampa() const {  
    cout << Giorno() << '/' << Mese() << '/' << Anno();  
}
```



La classe `dataora` eredita due metodi diversi con lo stesso nome e segnatura: `Stampa()` !

La seguente invocazione è segnalata dal compilatore come una illegalità dovuta ad **ambiguità**:

```
dataora d;  
d.Stampa(); // ILLEGALE
```

Ambiguity



Come al solito per gli errori di compilazione dovuti da ambiguità, la generazione dell'errore avviene **soltanto quando si tenta di invocare** tale metodo `Stampa()`



L'ambiguità rimarrebbe anche se le signature dei metodi nelle classi base fossero diverse

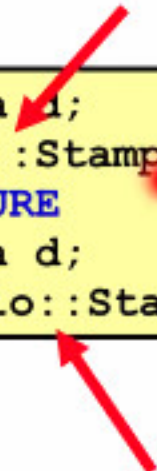
```
class A {
public:
    void f() {cout << "A::f ";}
};

class B {
public:
    void f(int x) {cout << "A::f ";}
};

class D: public A, public B {};

int main(){
    D d;
    d.f(); // Illegale: "request for member f is ambiguous"
    d.f(2); // Illegale: "request for member f is ambiguous"
}
```

Possiamo risolvere l'ambiguità usando l'operatore di scoping:



```
dataora d;  
d.data::Stampa();  
// OPPURE  
dataora d;  
d.orario::Stampa();
```

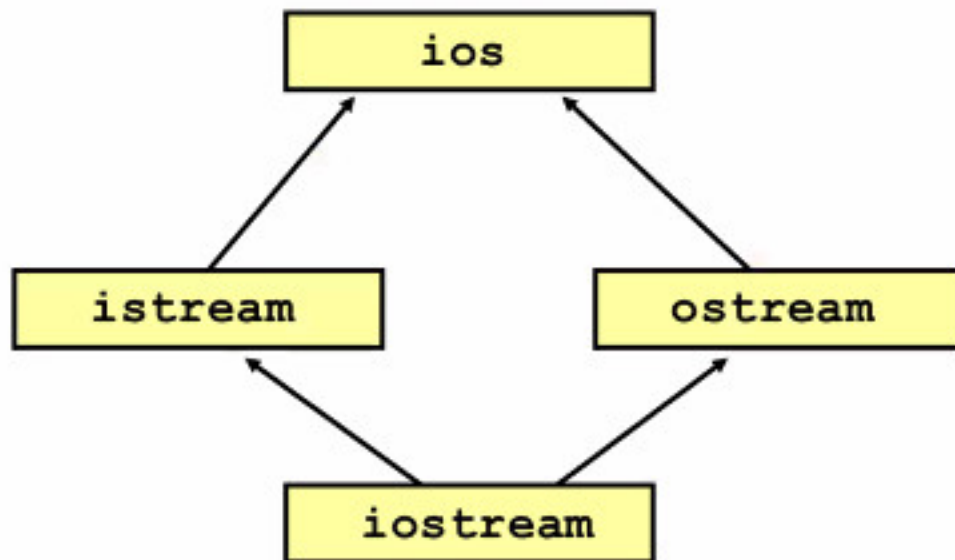


Una ridefinizione del metodo `Stampa()` nella classe `dataora` avrebbe nascosto entrambi i metodi `Stampa()` delle classi base e non ci sarebbe quindi stata alcuna ambiguità.

```
void dataora::Stampa() const {  
    data::Stampa(); cout << ' '; orario::Stampa();  
}
```


Esempio di ereditarietà multipla

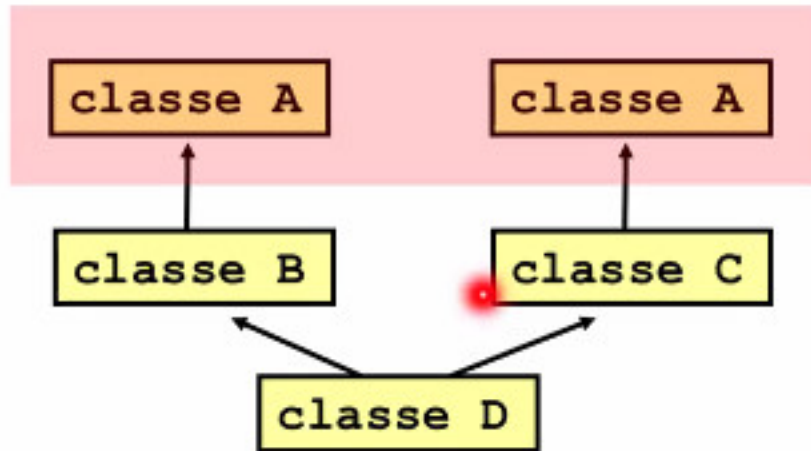
• **STREAM** •

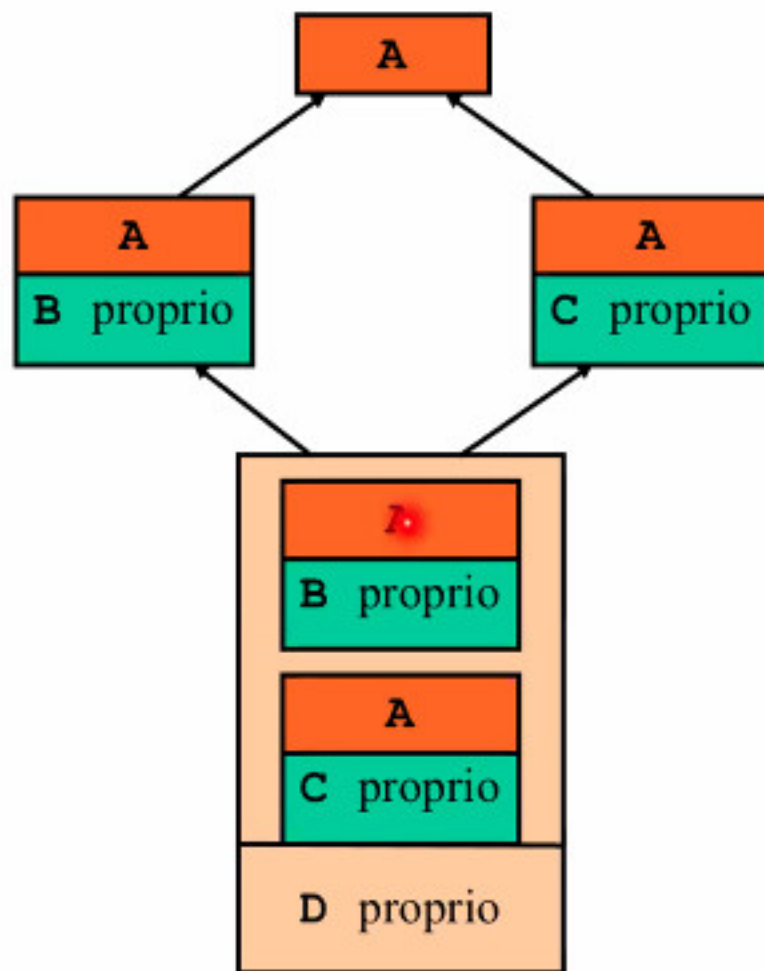


Ereditarietà a diamante



Stessa classe
base **A**





Due sottooggetti della classe base comune **A**:

- 1) ambiguità
- 2) o come minimo spreco di memoria



```
class A {  
public:  
    int a;  
    A(int x=1): a(x) {}  
};
```

```
class B: public A {  
public:  
    B(): A(2) {}  
};
```

```
class C: public A {  
public:  
    C(): A(3) {}  
};
```

```
class D: public B, public C { };
```

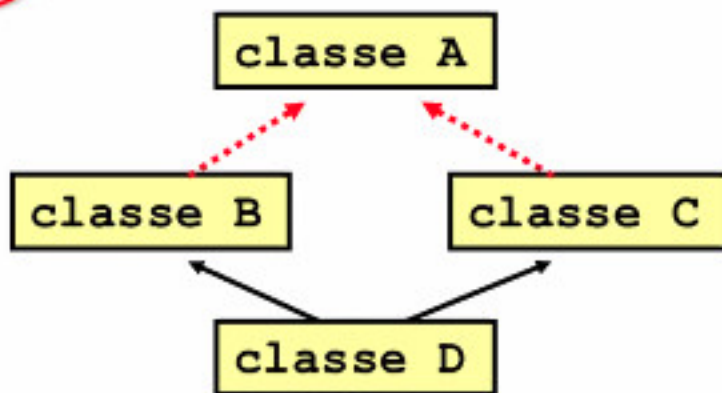
```
int main(){  
    D d;  
    A* p = &d;    // Illegale: A è una classe base ambigua per D  
    cout << p->a; // Quale sottooggetto di A si dovrebbe usare?  
}
```

```
class A {  
protected:  
    int x;  
public:  
    A(int y=0): x(y) {}  
    virtual void print() =0; // virtuale puro  
};  
class B: public A {  
public:  
    B(): A(1) {}  
    virtual void print() {cout << x;} // implementazione in B  
};  
class C: public A {  
public:  
    C(): A(2) {}  
    virtual void print() {cout << x;} // implementazione in C  
};  
  
class D: public B, public C {  
public:  
    virtual void print() {cout << x;} // overriding: quale x?  
};  
  
int main(){  
    D d;  
    A* p = &d; // ERRORE: "A is an ambiguous base of D"  
    p->print(); // la chiamata polimorfa non è legale  
}
```



Un **unico** sottooggetto di tipo **A**
in ogni oggetto della classe **D** che
chiude il diamante

*Derivazione
virtuale*




```
class A { // A è una classe base virtuale
```

```
...
```

```
};
```

```
class B : virtual public A {
```

```
...
```

```
};
```

```
class C : virtual public A {
```

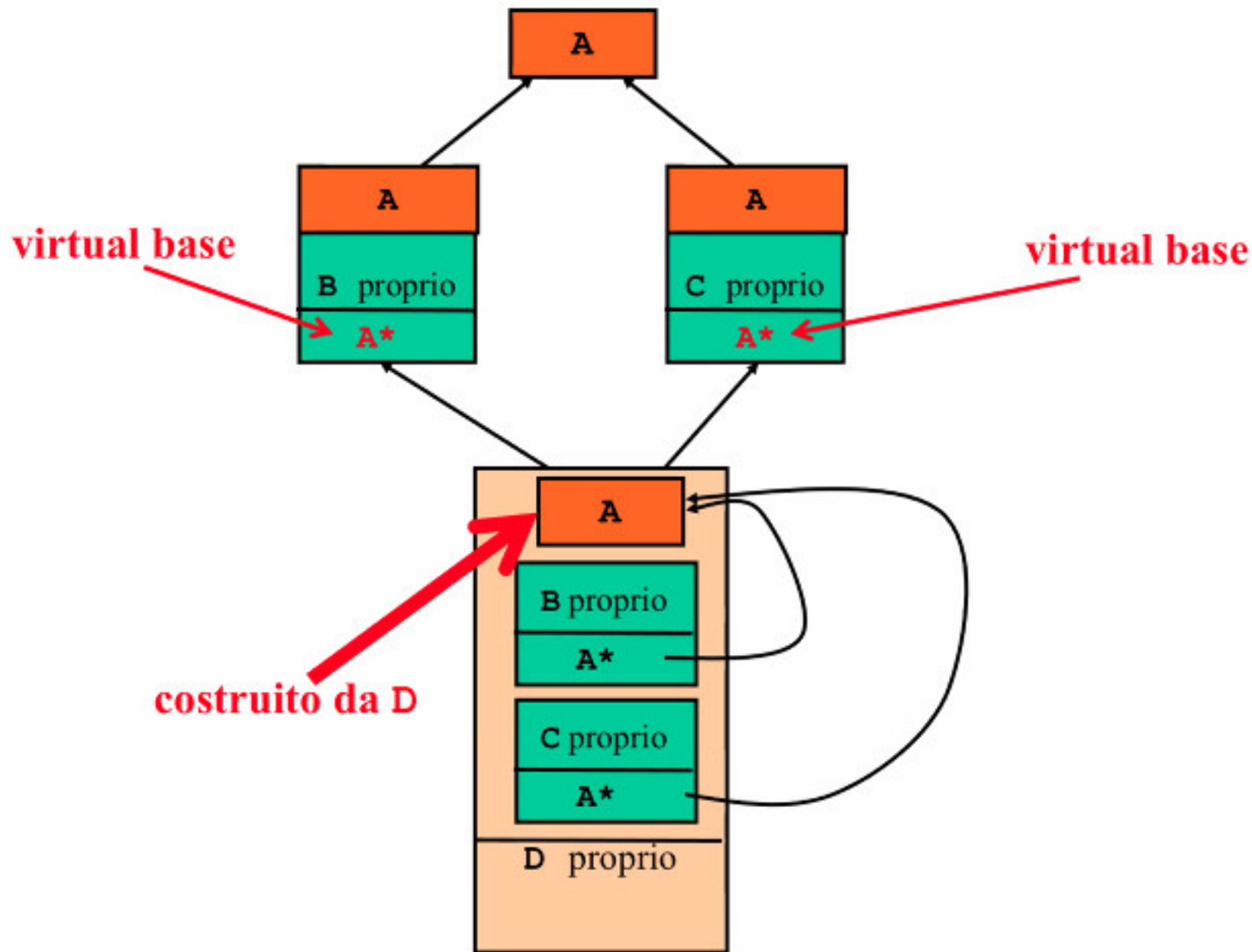
```
...
```

```
};
```

```
class D : public B, public C {
```

```
...
```

```
};
```



Examples

```
class A {  
    double d; // 8 byte  
};  
  
class B: public A {  
    // 8 byte  
};  
  
class C: public A {  
    // 8 byte  
};  
  
class D: public B, public C {  
};
```

```
class A {  
    double d; // 8 byte  
};  
  
class B: public A {  
    // 8+0 byte  
};  
  
class C: public A {  
    // 8+0 byte  
};  
  
class D: public B, public C {  
};  
  
int main() {  
    cout << "sizeof(A) == " << sizeof(A) << endl; // 8  
    cout << "sizeof(B) == " << sizeof(B) << endl; // 8=8+0  
    cout << "sizeof(C) == " << sizeof(C) << endl; // 8=8+0  
    cout << "sizeof(D) == " << sizeof(D) << endl; // 16=8+8  
}
```

```
class A {  
    double d; // 8 byte  
};  
  
class B: virtual public A {  
    // 8 byte + 1 puntatore (8 byte)  
};  
  
class C : virtual public A {  
    // 8 byte + 1 puntatore (8 byte)  
};  
  
class D: public B, public C {  
};  
  
int main() {  
    cout << "sizeof(A) == " << sizeof(A) << endl; // 8  
    cout << "sizeof(B) == " << sizeof(B) << endl; // 16=8+8  
    cout << "sizeof(C) == " << sizeof(C) << endl; // 16=8+8  
    cout << "sizeof(D) == " << sizeof(D) << endl; // 24=8+8+8  
}
```

Attenzione: rappresentazione non standard, i compilatori possono implementare in modo diverso



```
class A {
public:
    int a;
    A(int x=1): a(x) {}
};

class B: virtual public A {
public:
    B(): A(2) {}
};

class C: virtual public A {
public:
    C(): A(3) {}
};

class D: public B, public C {
};

int main(){
    D d;
    A* p = &d;    // compila
    cout << p->a; // stampa: 1 (e non 2 o 3!), perchè?
}
```



Unique final overrider rule

```
class A {
public:
    virtual void print()=0;
};

class B: virtual public A {
public:
    void print() override {cout << "B " ;}
};

class C: virtual public A {
public:
    void print() override {cout << "C " ;}
};

class D: public B, public C {
    // se ometto questo overriding si ottiene un errore di compilazione:
    // "no unique final overrider for A::print()"
    void print() override {cout << "D " ;}
};

int main(){
    D d;
    A* p = &d; // compila
    p->print(); // stampa: D
}
```

MOTIVAZIONE

La vtable di **D** deve avere un indirizzo per l'entry di **print()**

Unique final overrider rule

```
class A {
public:
    virtual void print() {cout << "A " ;}
};

class B: virtual public A {
public:
    // void print() override {cout << "B " ;}
};

class C: virtual public A {
public:
    // void print() override {cout << "C " ;}
};

class D: public B, public C {

    // in questo caso è legale, ereditiamo un "unique final overrider"
    // void print() override {cout << "D " ;}
};

int main(){
    D d;
    A* p = &d; // compila
    p->print(); // stampa: A
}
```


```
class A {
public:
    void print() {cout <<"A " ;}
};

class B: virtual public A {
public:
    void print() {cout << "B " ;}
};

class C: virtual public A {
public:
    void print() {cout << "C " ;}
};

class D: public B, public C {
    // eredito 2 metodi print() non virtuali,
    // compila ma ambiguità in compilazione per una invocazione d.print()
};

int main(){
    D d;
    A* p = &d;    // compila
    p->print();    // stampa: A
    d.print();    // Illegale: chiamata ambigua
    d.B::print(); // OK, stampa B
}
```



```
class A {  
public:  
    void print() {cout <<"A " ;}  
};
```

```
class B: virtual public A {  
public:  
    void print() {cout << "B " ;}  
};
```

```
class C: virtual public A {  
public:  
    void print() {cout << "C " ;}  
};
```

```
class D: public B, public C {  
    // eredito 2 metodi print() non virtuali,  
    // compila ma ambiguità per una invocazione d.print()  
};
```

```
// MOTIVAZIONE
```

```
// D non ha vtable, oppure non ha una entry per print() nella sua vtable
```

Anche per la derivazione virtuale multipla, possiamo avere derivazione privata, pubblica o protetta.

Vale la seguente **regola**: la derivazione protetta prevale su quella privata, e la derivazione pubblica prevale su quella protetta.

```
class A {  
public:  
    void f() {cout << "A";}  
};  
class B: virtual private A {}; // derivazione virtuale privata  
  
class C: virtual public A {}; // derivazione virtuale pubblica  
  
class D: public B, public C {}; // prevale la derivazione pubblica  
  
int main(){  
    D d;  
    d.f(); // OK, stampa: A (sarebbe C::f())  
    d.B::f(); // Illegale, non compila, A::f() inaccessibile  
}
```


Costruttore di D in presenza di basi virtuali

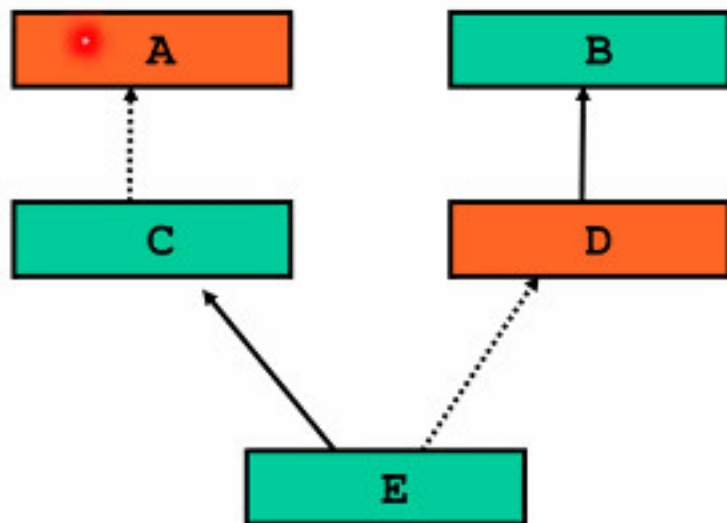


(1) Per primi vengono richiamati, una sola volta, i costruttori delle **classi base virtuali** che si trovano nella gerarchia di derivazione di D. Vi può essere più di una classe base virtuale nella gerarchia di derivazione di D: la ricerca delle classi base virtuali nella gerarchia procede seguendo l'ordine da sinistra verso destra e dall'alto verso il basso ("left-to-right top-down order").

(2) Una volta che sono stati invocati i costruttori delle classi virtuali nella gerarchia di derivazione di D, vengono richiamati i costruttori delle superclassi dirette non virtuali di D: questi costruttori **escludono di richiamare** eventuali costruttori di classi virtuali già richiamati al passo (1);

(3) Infine viene eseguito il costruttore "proprio" di D, ovvero vengono costruiti i campi dati propri di D e quindi viene eseguito il corpo del costruttore di D.

Le chiamate dei costruttori dei punti (1) e (2), **se non sono esplicite, vengono automaticamente inserite** dal compilatore nella lista di inizializzazione del costruttore di D: in questo caso, come al solito, si tratta di chiamate implicite ai costruttori di default.



```
class A {
public:
    A() { cout << "A ";}
};

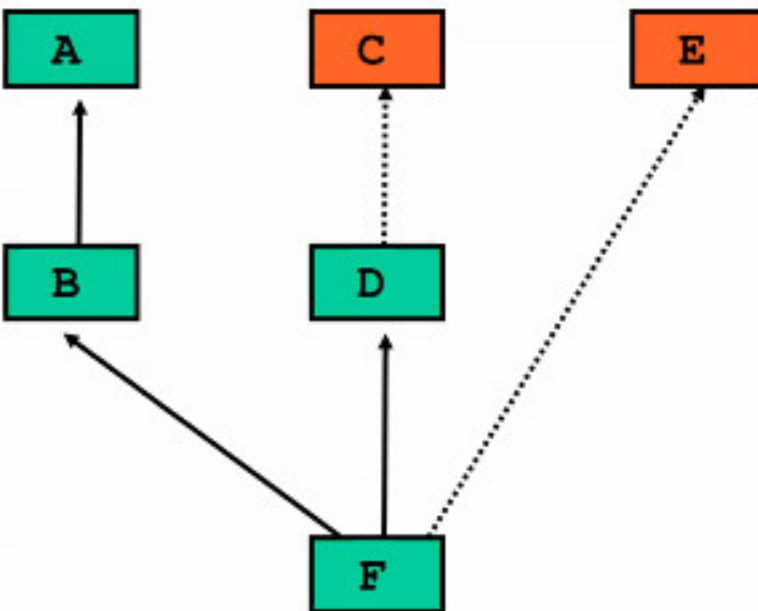
class B {
public:
    B() { cout << "B ";}
};

class C: virtual public A {
public:
    C(){ cout << "C ";}
};

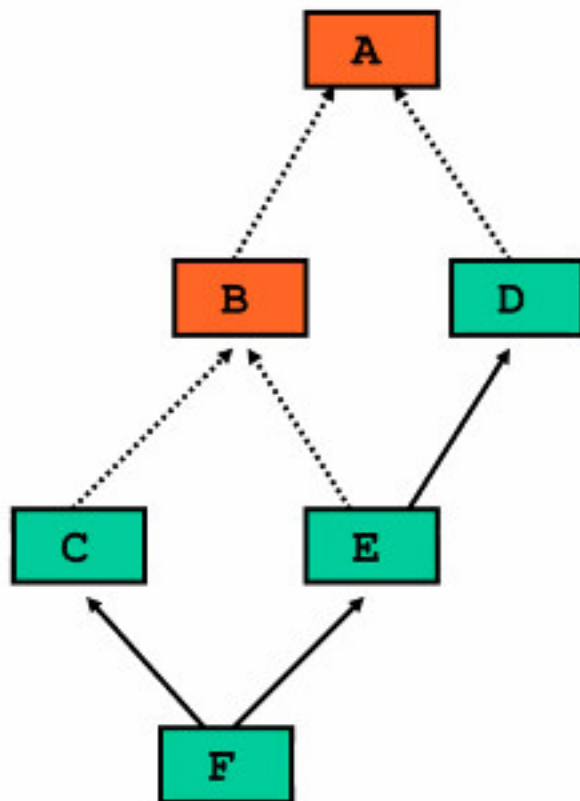
class D: public B {
public:
    D(){ cout << "D ";}
};

class E : public C, virtual public D {
public:
    E(){ cout << "E ";}
};

int main() {      E e; }
// stampa: A B D C E
```

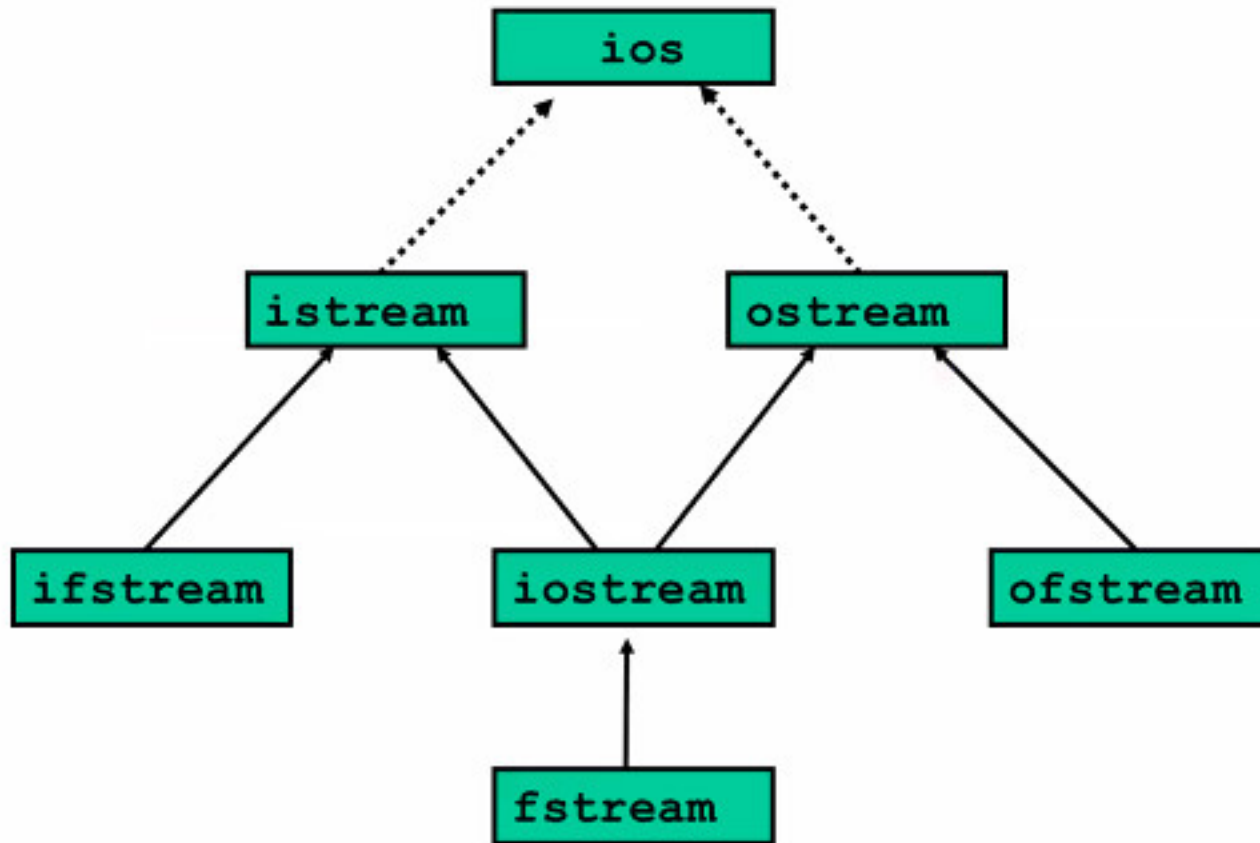


```
class A {
public:
    A() {cout << "A ";}
};
class B: public A {
public:
    B() {cout << "B ";}
};
class C {
public:
    C() {cout << "C ";}
};
class D: virtual public C {
public:
    D() {cout << "D ";}
};
class E {
public:
    E() {cout << "E ";}
};
class F:
    public B, public D, virtual public E
{
public:
    F() {cout << "F ";}
};
int main(){ F f; }
// stampa: C E A B D F
```

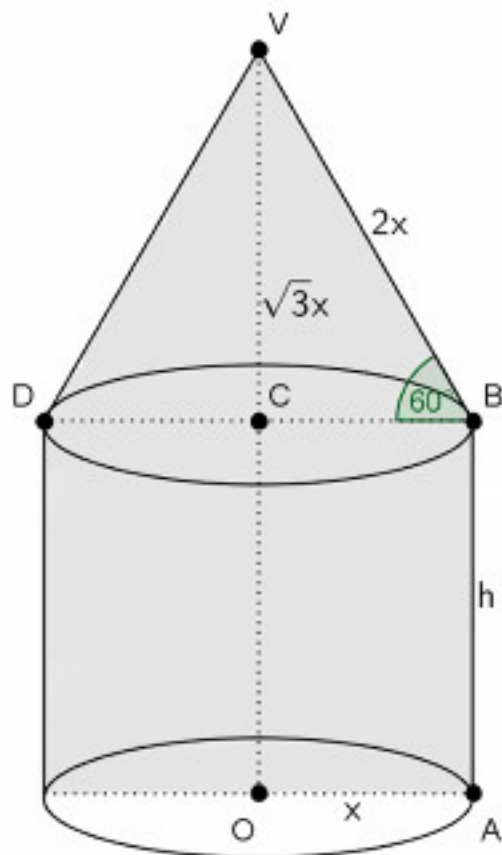


```
class A {  
public:  
    A() {cout << "A " ;}  
};  
class B: virtual public A {  
public:  
    B() {cout << "B " ;}  
};  
class D: virtual public A {  
public:  
    D() {cout << "D " ;}  
};  
class C: virtual public B {  
public:  
    C() {cout << "C " ;}  
};  
class E: virtual public B, public D {  
public:  
    E() {cout << "E " ;}  
};  
class F: public C, public E {  
public:  
    F() {cout << "F " ;}  
};  
int main(){ F f; }  
// stampa: A B C D E F
```

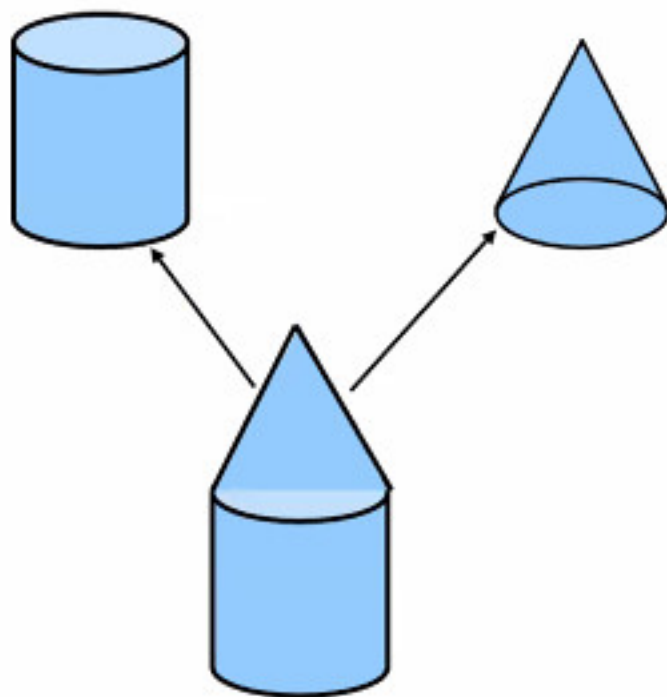
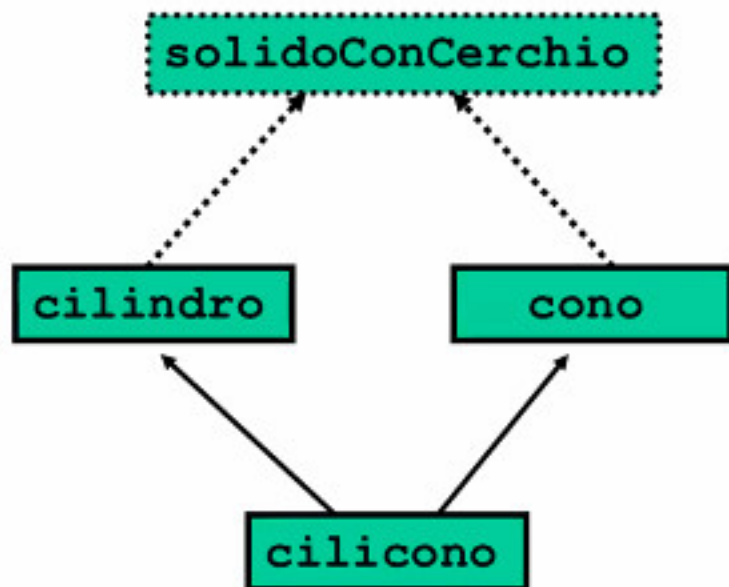
La gerarchia delle classi di input/output



Esempio del missile



Esempio del missile



```
#include <cmath>

// gli oggetti sono solidi che hanno un cerchio come faccia

class solidoConCerchio { // classe base astratta virtuale
protected:
    double raggio; // del cerchio
    double circonferenza() const {return (2*M_PI*raggio);}
    double area_cerchio() const {return (M_PI*raggio*raggio);}
public:
    // no costruttore di default
    solidoConCerchio(double r): raggio(r) {}
    // virtuale pura: area del solido che ha un cerchio come faccia
    virtual double area() const = 0;
    // virtuale pura: volume del solido che ha un cerchio come faccia
    virtual double volume() const = 0;
};
```

```
// derivazione virtuale
class cilindro: virtual public solidoConCerchio {
protected:
    double altezza; // altezza del cilindro
    double area_laterale() const {
        return (circonferenza()*altezza);
    }
public:
    cilindro(double r, double h): solidoConCerchio(r), altezza(h) {}
    double area() const override {
        return (2*area_cerchio() + area_laterale());
    }
    double volume() const override {
        return (area_cerchio()*altezza);
    }
};
```

```

// derivazione virtuale
class cono: virtual public solidoConCerchio {
protected:
    double altezza; // altezza del cono
    double area_laterale() const {
        double apotema = sqrt(raggio*raggio + altezza*altezza);
        return (2*circonferenza()*apotema);
    }
public:
    cono(double r, double h): solidoConCerchio(r), altezza(h) {}
    double area() const override {
        return (area_cerchio() + area_laterale());
    }
    double volume() const override {
        return (area_cerchio()*altezza/3);
    }
};

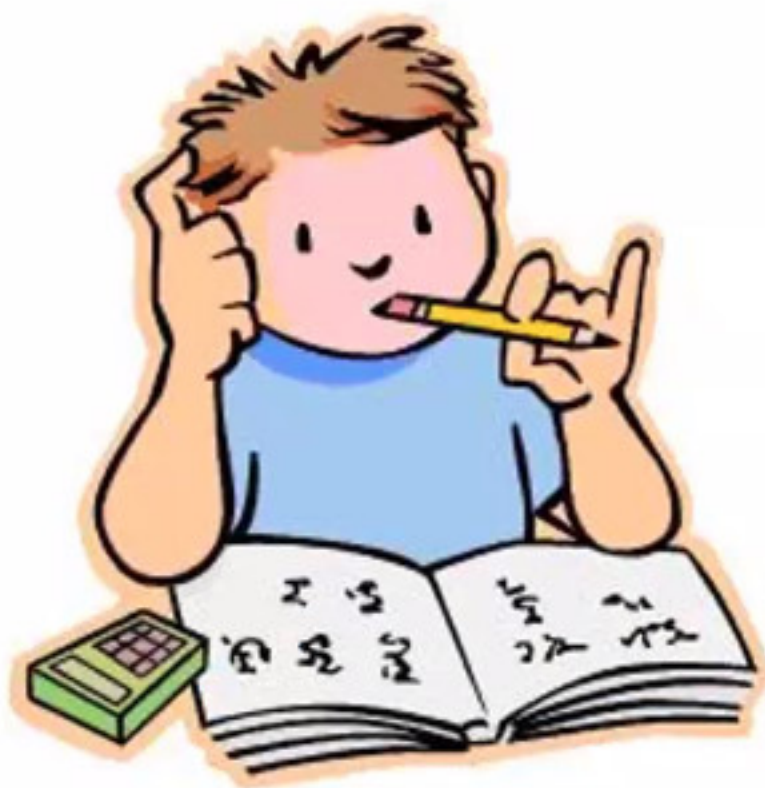
```

parametri ignorati

```
class cilicono: public cilindro, public cono {  
    // derivazione multipla: un solo sottooggetto solidoConCerchio  
public:  
    cilicono(double r, double h1, double h2) :  
        solidoConCerchio(r), cilindro(r, h1), cono(r, h2) {}  
    // NOTA BENE: senza l'invocazione esplicita solidoConcerchio(r)  
    // non compilerebbe perchè non esiste il costruttore di default  
    // solidoConCerchio()  
  
    double area() const override {  
        return (cilindro::area_laterale() +  
                cono::area_laterale() + area_cerchio());  
    }  
    // NOTA BENE: eredito 2 metodi area_laterale() e volume()  
    // necessario l'uso dell'operatore di scoping  
    double volume() const override {  
        return (cilindro::volume() + cono::volume());  
    }  
};
```



```
int main() {  
    solidoConCerchio* p;  
    cilindro cil(1, 2); cono co(1, 2);  
    cilicono clc(1, 2, 2);  
    p = &cil;  
    cout << "Area cilindro: " << p->area() << endl;  
    cout << "Volume cilindro: " << p->volume() << endl;  
    p = &co;  
    cout << "Area cono: " << p->area() << endl;  
    cout << "Volume cono: " << p->volume() << endl;  
    p = &clc;  
    cout << "Area cilicono: " << p->area() << endl;  
    cout << "Volume cilicono: " << p->volume() << endl;  
}
```

```
class Z {
public:
    Z() {cout << "Z() ";}
    Z(const Z& x) {cout << "Zc ";}
};

class A {
private:
    Z w;
public:
    A() {cout << "A() ";}
    A(const A& x) {cout << "Ac ";}
};

class B: virtual public A {
private:
    Z z;
public:
    B() {cout << "B() ";}
    B(const B& x) {cout << "Bc ";}
};

class C: virtual public A {
private:
    Z z;
public:
    C() {cout << "C() ";}
};

class D: public B, public C {
public:
    D() {cout << "D() ";}
    D(const D& x): C(x) {cout << "Dc ";}
};
```

Si assuma che A, B, C, D siano quattro classi polimorfe. Si consideri il seguente `main()`.

```
main() {  
    A a; B b; C c; D d;  
    cout << (dynamic_cast<D*>(&c) ? "0 " : "1 ");  
    cout << (dynamic_cast<B*>(&c) ? "2 " : "3 ");  
    cout << (!dynamic_cast<C*>(&b)) ? "4 " : "5 ";  
    cout << (dynamic_cast<B*>(&a) || dynamic_cast<C*>(&a) ? "6 " : "7 ");  
    cout << (dynamic_cast<D*>(&b) ? "8 " : "9 ");  
}
```

Si supponga che tale `main()` compili ed esegua correttamente. Disegnare i diagrammi di **tutte** le possibili gerarchie per le classi A, B, C, D tali che l'esecuzione del `main()` provochi la stampa: 0 3 4 6 8.