

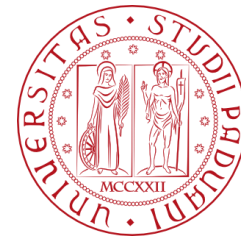
# Tutorato 6

06/12/2023

Programmazione ad Oggetti – 2023-2024

Gabriel Rovesti

2103389 – LM Computer Science



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA



# Ereditarietà multipla



```
1  #ifndef DATAORA_H
2  #define DATAORA_H
3  #include "orario.h"
4  class dataora : public orario, public data
5  {
6  public:
7  dataora() (int a=1, int b=1, int c=1970, int d=0, int e=0, int f=0) :
8  orario(d,e,f), data(a,b,c) {}
```

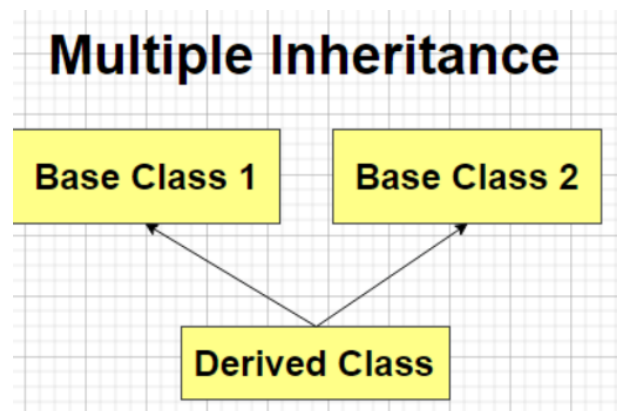


```
1  void orario::Stampa() const{
2      cout << ore << ":" << minuti << ":" << secondi << endl;
3  }
4
5  void data::Stampa() const{
6      cout << giorno << "/" << mese << "/" << anno << endl;
7  }
8
9  // classe dataora con due diversi metodi Stampa()
10 dataora d;
11 d.stampa(); // illegale: quale chiamerebbe?
```



- Se anche le signature (firme) dei metodi fossero diverse, il problema rimarrebbe
- Possiamo usare l'operatore di scoping per risolvere l'ambiguità
  - Chiamando quella di orario oppure di dataora
- Una ridefinizione opportuna nella classe derivata (dataora) avrebbe nascosto entrambi i metodi delle classi basi e non ci sarebbero state ambiguità

Siamo quindi in un caso di ereditarietà multipla.



# Ereditarietà multipla

- I costruttori delle superclassi sono chiamati nell'ordine di derivazione, partendo da destra e andando verso sinistra oppure seguendo l'ordine innestato

Ci possono essere dei problemi:

- due sottooggetti di una classe comune portano ad ambiguità
- non si sa esattamente quale oggetto chiamare

```
1  class base1 {
2      public:
3          void someFunction( ) {....}
4  };
5  class base2 {
6      void someFunction( ) {....}
7  };
8  class derived : public base1, public base2 {};
9
10 int main() {
11     derived obj;
12     obj.someFunction() // Error!
13 }
```

# Ereditarietà multipla



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

```
1  //Esempio errore ereditarietà multipla
2
3  #include <iostream>
4  using namespace std;
5
6  class A {
7      protected:
8          int x;
9  public:
10     A(int y) : x(y){}
11     virtual void print() =0;
12 };
13
14 class B { // implementazione in B
15 public:
16     B(): A(1) {}
17     virtual void print() { cout << "B" << endl; }
18 };
19
```



# Ereditarietà multipla



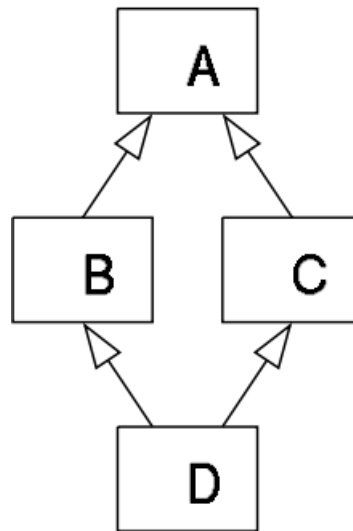
UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

```
1 class C : public A, public B { // implementazione in C (chi chiama?)
2 public:
3     C(): A(2) {}
4     virtual void print() { cout << "C" << endl; }
5 };
6
7 int main() {
8     C c;
9     A* p = &c;
10    p->print(); // la chiamata è ambigua
11    // Errore: "A is an ambiguous base of C"
12 }
```



Accade il problema del diamante/diamond problem:

- Quando ereditiamo più di una classe base nella stessa classe derivata e tutte queste classi base ereditano da una stessa classe padre, (superclasse), più riferimenti della superclasse diventano disponibili per la classe derivata.
- Quindi, non è chiaro alla classe derivata a quale versione della classe super genitore debba fare riferimento.



# Ereditarietà multipla



```
1  class A{}; // A classe base virtuale
2
3  class B : virtual public A{}; // B eredita virtualmente da
4
5  class C : virtual public A{}; // C eredita virtualmente da A
6
7  class D : public B, public C{}; // D eredita da B e C
```

La soluzione al problema del diamante è l'uso della parola chiave virtual.

Trasformiamo le due classi genitore (che ereditano dalla stessa superclasse) in sottoclassi virtuali, in modo da evitare due copie della classe superclasse nella classe figlio. In questo modo, solo un'istanza delle classi verrà invocata, evitando *ambiguità*.





# Ereditarietà multipla



```
1  class A{}; // A classe base virtuale
2
3  class B : virtual public A{}; // B eredita virtualmente da
4
5  class C : virtual public A{}; // C eredita virtualmente da A
6
7  class D : public B, public C{}; // D eredita da B e C
```

La soluzione al problema del diamante è l'uso della parola chiave virtual.

Trasformiamo le due classi genitore (che ereditano dalla stessa superclasse) in sottoclassi virtuali, in modo da evitare due copie della classe superclasse nella classe figlio. In questo modo, solo un'istanza delle classi verrà invocata, evitando *ambiguità*.



# Ereditarietà multipla



```
1  #include<iostream>
2  using namespace std;
3  class Person { //class Person
4  public:
5      Person() { cout << "Person::Person() called" << endl; } //Base constructor
6      Person(int x) { cout << "Person::Person(int) called" << endl; }
7  };
8
9  class Father : virtual public Person { //class Father inherits Person
10 public:
11     Father(int x):Person(x) {
12         cout << "Father::Father(int) called" << endl;
13     }
14 };
15
16 class Mother : virtual public Person { //class Mother inherits Person
17 public:
18     Mother(int x):Person(x) {
19         cout << "Mother::Mother(int) called" << endl;
20     }
21 };
22
```



# Ereditarietà multipla



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

```
1  class Child : public Father, public Mother { //class Child inherits Father and Mother
2  public:
3      Child(int x):Mother(x), Father(x) {
4          cout << "Child::Child(int) called" << endl;
5      }
6  };
7
8  int main() {
9      Child child(30);
10 }
11
12 /*
13 Person::Person() called
14 Father::Father(int) called
15 Mother::Mother(int) called
16 Child::Child(int) called
17 */
```



# Ereditarietà multipla



L'ereditarietà multipla richiede ordine nella ridefinizione dei metodi.

Ricordiamo che **override** serve a ridefinire un comportamento nelle classi derivate.

Per ordine di ereditarietà, conviene sempre specificare «dove finisce l'overriding», rendendo chiaro al compilatore quale classe chiamare.

```
1  class A{
2      public:
3          virtual void print() { cout << "A" << endl; }
4  }
5
6  class B : virtual public A{
7      public:
8          void print() override { cout << "B" << endl; }
9  }
10
11 class C : virtual public A{
12     public:
13         void print() override { cout << "C" << endl; }
14 }
15
16 class D : public B, public C{
17     public:
18         void print() override { cout << "D" << endl; }
19 }
```



# Ereditarietà multipla



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

```
class D : public B, public C{
    public:
    void print() override { cout << "D" << endl; }
}
// in questo caso, tutti stanno facendo ridefinire il metodo verso il basso
// quindi, se si chiama print() su un oggetto di tipo D, si avrà A come output

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



# Ereditarietà multipla



```
1  class A{
2      public:
3          virtual void print() { cout << "A" << endl; }
4  }
5
6  class B : virtual public A{
7      public:
8          void print() { cout << "B" << endl; }
9  }
10
11 class C : virtual public A{
12     public:
13         void print() { cout << "C" << endl; }
14 }
15
16 class D : public B, public C{
17     //se non marco virtual,
18     //error: request for member 'print' is ambiguous
19     //compila, ma rimane ambiguo su chi chiamare
20 };
21
```



1. Per primi vengono richiamati i costruttori delle classi basi virtuali
  - In presenza di più basi virtuali si segue l'ordine da sinistra verso destra e dall'alto verso il basso
2. Vengono poi richiamati i costruttori delle superclassi dirette non virtuali di D
  - Questi escludono di richiamare costruttori virtuali già richiamati al passo (1)
3. Viene chiamato infine il costruttore proprio della classe derivata
  - Vengono costruiti i campi dati propri della derivata e poi eseguito il corpo del costruttore

# Esempio 1



```
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 ";}
};
```

```
D d1;
D d2 = d1;
```





# Esempio 1: Soluzione

```
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 ";}
};
```


```
D d1;
D d2 = d1;
```

```
1 // Z() A() Z() B() Z() C() D() - esempio 1
2 // Z() A() Z() B() Zc Dc - esempio 2
```

# Esempio 2



```
3  #include <iostream>
4  using namespace std;
5
6  class A {
7  public:
8      A() { cout << "A" << endl; }
9  };
10
11 class B {
12 public:
13     B() { cout << "B" << endl; }
14 };
15
16 class C: virtual public A{
17 public:
18     C() { cout << "C" << endl; }
19 };
```



```
1  class D: virtual public B{
2  public:
3      D() { cout << "D" << endl; }
4  };
5
6  class E: public C, virtual public D{
7  public:
8      E() { cout << "E" << endl; }
9  };
10
11 int main() {
12     E e;
13     return 0;
14 }
```



# Esempio 2: Soluzione

```
3  #include <iostream>
4  using namespace std;
5
6  class A {
7  public:
8      A() { cout << "A" << endl; }
9  };
10
11 class B {
12 public:
13     B() { cout << "B" << endl; }
14 };
15
16 class C: virtual public A{
17 public:
18     C() { cout << "C" << endl; }
19 };
```

```
1  class D: virtual public B{
2  public:
3      D() { cout << "D" << endl; }
4  };
5
6  class E: public C, virtual public D{
7  public:
8      E() { cout << "E" << endl; }
9  };
10
11 int main() {
12     E e;
13     return 0;
14 }
```

```
1  // Output:
2  // A B D C E
```

# Esempio 3



```
1 class A {
2 public:
3     A() { cout << "A" << endl; }
4 };
5
6 class B: public A {
7 public:
8     B() { cout << "B" << endl; }
9 };
10
11 class C{
12 public:
13     C() { cout << "C" << endl; }
14 };
15
16 class D: virtual public C{
17 public:
18     D() { cout << "D" << endl; }
19 };
```

```
1 class E{
2 public:
3     E() { cout << "E" << endl; }
4 };
5
6 class F: public B, public D, virtual public E {
7 public:
8     F() { cout << "F" << endl; }
9 };
10
11 int main() {
12     F f;
13     return 0;
14 }
15
```



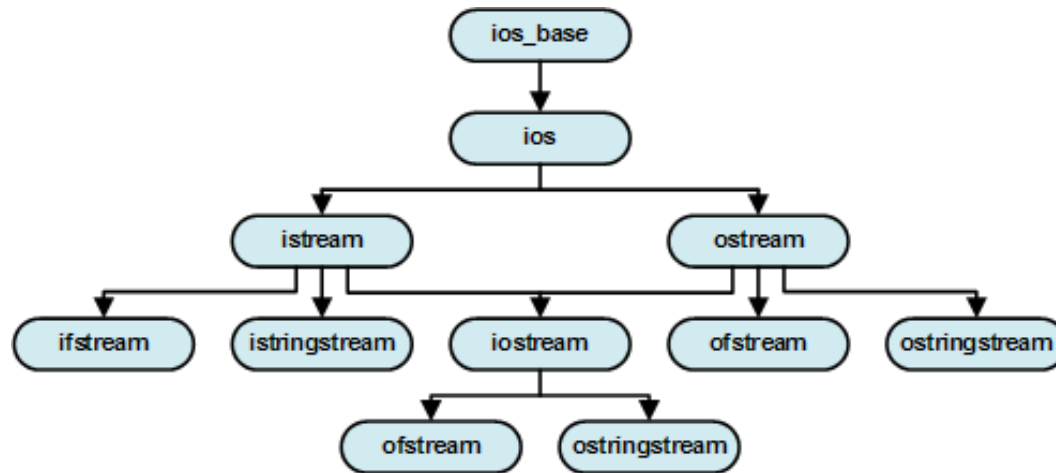
# Esempio 3: Soluzione

```
1 class A {
2 public:
3     A() { cout << "A" << endl; }
4 };
5
6 class B: public A {
7 public:
8     B() { cout << "B" << endl; }
9 };
10
11 class C{
12 public:
13     C() { cout << "C" << endl; }
14 };
15
16 class D: virtual public C{
17 public:
18     D() { cout << "D" << endl; }
19 };
```

```
1 class E{
2 public:
3     E() { cout << "E" << endl; }
4 };
5
6 class F: public B, public D, virtual public E {
7 public:
8     F() { cout << "F" << endl; }
9 };
10
11 int main() {
12     F f;
13     return 0;
14 }
15
```

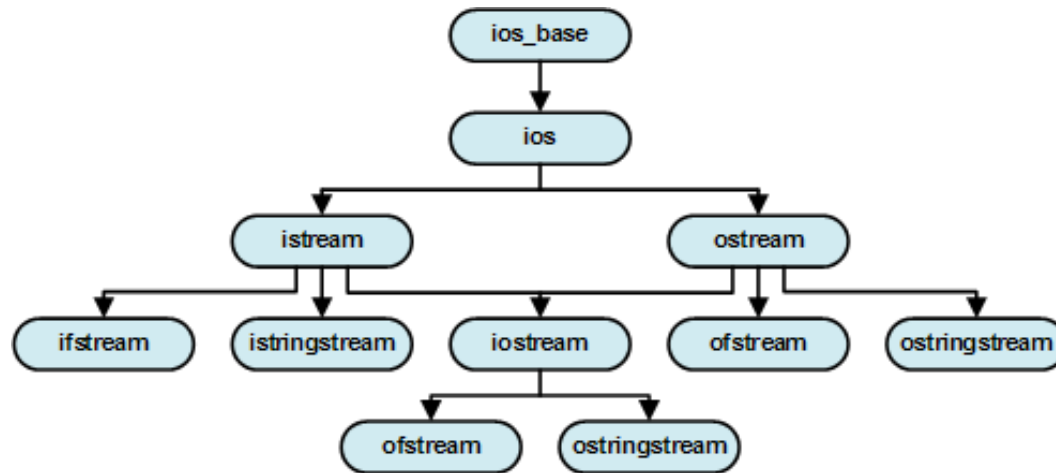
```
1 // Output:
2 // C E A B D F
```

# Gerarchia classi I/O



- **`istream`** ed **`ostream`** sono derivati virtualmente da `ios`
- **`istream`** effettua l'overloading dei membri di input per tipi primitivi e caratteri (parsing dei dati, con operatori come `>>`)
  - Prevede metodi come **`get()`**, **`ignore()`**, **`read()`** ed errori come **`failbit`**
- **`ostream`** rappresenta gli stream di output e comprende membri come **`cout`** o **`cerr`**
  - Prevede metodi come **`put()`**, **`write()`**

# Gerarchia classi I/O



- Gli stream di file sono `ifstream`, `ofstream` e `fstream`
- Gli stream associati a stringhe sono `istringstream`, `ostringstream` e `stringstream`

- Si tratta di gestire dei comportamenti anomali rispetto al normale comportamento del programma, salvando lo stato dell'esecuzione e lasciandolo eseguire ad una specifica subroutine (procedura)
  - La funzione lancia/sollewa un'eccezione (**throw**)
  - Nella funzione chiamante, viene controllato il blocco di cui gestire l'eccezione (**try**) e poi gestita l'eccezione (**catch**)

```
1 class Eccezione{
2 public:
3     Eccezione(string s): msg(s) {};;
4 };
5
6 telefonata bolletta::get(int i) const{
7     if(i<0 || i>=num) throw Eccezione("Indice non valido");
8     return v[i];
9 }
```

```
1 int main(){
2     try{
3         bolletta b;
4         telefonata t = b.get(10);
5     }catch(Eccezione e){
6         cout << e.msg << endl;
7     }
8 }
```



# Esercizio 1: Funzione

## Esercizio Funzione

Si richiamano i seguenti fatti concernenti la libreria di I/O standard.

- `ios` è la classe base astratta e virtuale della gerarchia di tipi della libreria di I/O; la classe `istream` è derivata direttamente e virtualmente da `ios`; la classe `ifstream` è derivata direttamente da `istream`.
- `ios` rende disponibile un metodo costante e non virtuale `bool fail()` con il seguente comportamento: una invocazione `s.fail()` ritorna `true` se e solamente se lo stream `s` è in uno stato di fallimento (cioè, il failbit di `s` vale 1).
- `istream` rende disponibile un metodo non costante e non virtuale `long tellg()` con il seguente comportamento: una invocazione `s.tellg()`:
  1. se `s` è in uno stato di fallimento allora ritorna -1;
  2. altrimenti, cioè se `s` non è in uno stato di fallimento, ritorna la posizione della testina di input di `s`.
- `ifstream` rende disponibile un metodo costante e non virtuale `bool is_open()` con il seguente comportamento: una invocazione `s.is_open()` ritorna `true` se e solo se il file associato allo stream `s` è aperto.

Definire una funzione `long Fun(const ios&)` con il seguente comportamento: una invocazione `Fun(s)`:

- (1) se `s` è in uno stato di fallimento lancia una eccezione di tipo `Fallimento`, dove la classe `Fallimento` va esplicitamente definita;
- (2) se `s` non è in uno stato di fallimento allora:
  - (a) se `s` non è un `ifstream` ritorna -2;
  - (b) se `s` è un `ifstream` ed il file associato non è aperto ritorna -1;
  - (c) se `s` è un `ifstream` ed il file associato è aperto ritorna la posizione della cella corrente di input di `s`.



# Esercizio 1: Soluzione



```
1  class Fallimento{
2      private:
3          string errore;
4      public:
5          Fallimento(string e): errore(e) {}
6  };
7
8  long Fun(const ios& s){
9      if(s.fail()) throw Fallimento("Errore");
10
11     ifstream* i=dynamic_cast<ifstream*>(const_cast<ios*>(&s));
12     if(!i) return -2;
13     if(!i && !i->is_open()) return -1;
14     if(i && i->is_open()) return i->tellg();
15     return 0;
16 }
```



# Esercizio 2: Compila/Non compila

```
class A {
    bool x;
public:
    virtual ~A() = default;
};

class B {
    bool y;
public:
    virtual void f() const { cout << "B::f "; }
};

class C: public A {};

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

class E: public D {
public:
    void f() const { cout << "E::f "; }
};

template<class T>
void Fun(const T& ref) {
    try{ throw ref; }
    catch(const C& c) {cout << "C ";}
    catch(const E& e) {cout << "E "; e.f();}
    catch(const B& b) {cout << "B "; b.f();}
    catch(const A& a) {cout << "A ";}
    catch(const D& d) {cout << "D ";}
    catch(...)        {cout << "GEN ";}
}
```

Le precedenti definizioni compilano senza provocare errori (con gli opportuni `#include` e `using`). Per ognuna delle seguenti istruzioni di invocazione della funzione `Fun` scrivere nell'apposito spazio:

- **NON COMPILA** se la compilazione dell'istruzione provoca un errore;
- **ERRORE RUN-TIME** se l'istruzione compila correttamente ma la sua esecuzione provoca un errore a run-time;
- se l'istruzione compila correttamente e non provoca errori a run-time allora si scriva la stampa che l'esecuzione produce in output su `cout`; se non provoca alcuna stampa allora si scriva **NESSUNA STAMPA**.

# Esercizio 2: Compila/Non compila



```
class A {  
    bool x;  
public:  
    virtual ~A() = default;  
};
```

```
class B {  
    bool y;  
public:  
    virtual void f() const { cout << "B::f "; }  
};
```

```
class C: public A {};
```

```
class D: public B {  
public:  
    void f() const { cout << "D::f "; }  
};
```

```
class E: public D {  
public:  
    void f() const { cout << "E::f "; }  
};
```

```
template<class T>  
void Fun(const T& ref) {  
    try{ throw ref; }  
    catch(const C& c) {cout << "C ";}  
    catch(const E& e) {cout << "E "; e.f();}  
    catch(const B& b) {cout << "B "; b.f();}  
    catch(const A& a) {cout << "A ";}  
    catch(const D& d) {cout << "D ";}  
    catch(...)        {cout << "GEN ";}  
}
```

---

```
Fun(c);
```

---

```
Fun(d);
```

---

```
Fun(e);
```

---

```
Fun(a1);
```

---

```
Fun(b1);
```

---

```
Fun(d1);
```

---

```
Fun(*pd);
```

---

```
Fun<D>(*pd);
```

---

```
Fun<D>(e);
```

---

```
Fun<E>(*pd);
```

---

```
Fun<E>(e);
```

---

```
Fun<E>(d1);
```

---

```
Fun<A>(c);
```

```
C c; D d; E e; A& a1 = c; B& b1 = d; B& b2 = e; D& d1 = e; D* pd = dynamic_cast<E*>(&b2);
```



# Esercizio 2: Soluzione



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

```
● ● ●  
  
1  C  
2  B D::f  
3  E E::f  
4  A  
5  B D::f oppure B B::f  
6  B E::f oppure B D::f  
7  B E::f oppure B D::f  
8  B E::f oppure B D::f  
9  B E::f oppure B D::f  
10 NON COMPILA  
11 E E::f  
12 NON COMPILA  
13 A
```



# Esercizio 3: Cosa Stampa

```
class B {
public:
    B() {cout<< " B() ";}
    virtual ~B() {cout<< " ~B() ";}
    virtual void f() {cout <<" B::f "; g(); j();}
    virtual void g() const {cout <<" B::g ";}
    virtual const B* j() {cout<<" B::j "; return this;}
    virtual void k() {cout <<" B::k "; j(); m(); }
    void m() {cout <<" B::m "; g(); j();}
    virtual B& n() {cout <<" B::n "; return *this;}
};
```

```
class C: virtual public B {
public:
    C() {cout<< " C() ";}
    ~C() {cout<< " ~C() ";}
    virtual void g() const override {cout <<" C::g ";}
    void k() override {cout <<" C::k "; B::n();}
    virtual void m() {cout <<" C::m "; g(); j();}
    B& n() override {cout <<" C::n "; return *this;}
};
```

```
class E: public C, public D {
public:
    E() {cout<< " E() ";}
    ~E() {cout<< " ~E() ";}
    virtual void g() const {cout <<" E::g ";}
    const E* j() {cout <<" E::j "; return this;}
    void m() {cout <<" E::m "; g(); j();}
    D& n() final {cout <<" E::n "; return *this;}
};
```

```
B* p1 = new E(); B* p2 = new C(); B* p3 = new D(); C* p4 = new E();
const B* p5 = new D(); const B* p6 = new E(); const B* p7 = new F(); F f;
```

```
class D: virtual public B {
public:
    D() {cout<< " D() ";}
    ~D() {cout<< " ~D() ";}
    virtual void g() {cout <<" D::g ";}
    const B* j() {cout <<" D::j "; return this;}
    void k() const {cout <<" D::k "; k();}
    void m() {cout <<" D::m "; g(); j();}
};
```

```
class F: public E {
public:
    F() {cout<< " F() ";}
    ~F() {cout<< " ~F() ";}
    F(const F& x): B(x) {cout<< " Fc ";}
    void k() {cout <<" F::k "; g();}
    void m() {cout <<" F::m "; j();}
};
```

# Esercizio 3: Cosa Stampa

```
class B {
public:
    B() {cout<< " B() ";}
    virtual ~B() {cout<< " ~B() ";}
    virtual void f() {cout <<" B::f "; g(); j();}
    virtual void g() const {cout <<" B::g ";}
    virtual const B* j() {cout<<" B::j "; return this;}
    virtual void k() {cout <<" B::k "; j(); m(); }
    void m() {cout <<" B::m "; g(); j();}
    virtual B& n() {cout <<" B::n "; return *this;}
};
```

```
class C: virtual public B {
public:
    C() {cout<< " C() ";}
    ~C() {cout<< " ~C() ";}
    virtual void g() const override {cout <<" C::g ";}
    void k() override {cout <<" C::k "; B::n();}
    virtual void m() {cout <<" C::m "; g(); j();}
    B& n() override {cout <<" C::n "; return *this;}
};
```

```
class E: public C, public D {
public:
    E() {cout<< " E() ";}
    ~E() {cout<< " ~E() ";}
    virtual void g() const {cout <<" E::g ";}
    const E* j() {cout <<" E::j "; return this;}
    void m() {cout <<" E::m "; g(); j();}
    D& n() final {cout <<" E::n "; return *this;}
};
```

```
B* p1 = new E(); B* p2 = new C(); B* p3 = new D(); C* p4 = new E();
const B* p5 = new D(); const B* p6 = new E(); const B* p7 = new F(); F f;
```

```
class D: virtual public B {
public:
    D() {cout<< " D() ";}
    ~D() {cout<< " ~D() ";}
    virtual void g() {cout <<" D::g ";}
    const B* j() {cout <<" D::j "; return this;}
    void k() const {cout <<" D::k "; k();}
    void m() {cout <<" D::m "; g(); j();}
};
```

```
class F: public E {
public:
    F() {cout<< " F() ";}
    ~F() {cout<< " ~F() ";}
    F(const F& x): B(x) {cout<< " Fc ";}
    void k() {cout <<" F::k "; g();}
    void m() {cout <<" F::m "; j();}
};
```

# Esercizio 3: Cosa Stampa

```
F x; .....  
  
C* p = new F(f); .....  
  
p1->f(); .....  
  
p1->m(); .....  
  
(p1->j())->k(); .....  
  
(dynamic_cast<const F*>(p1->j()))->g();  
  
p2->f(); .....  
  
p2->m(); .....  
  
(p2->j())->g(); .....  
  
p3->f(); .....  
  
p3->k(); .....  
  
(p3->n()).m(); .....  
  
(dynamic_cast<D&>(p3->n())).g(); .....
```

```
p4->f(); .....  
  
p4->k(); .....  
  
(p4->n()).m(); .....  
  
(p5->n()).g(); .....  
  
(dynamic_cast<E*>(p6))->j(); .....  
  
(dynamic_cast<C*>(const_cast<B*>(p7)))->k(); .....  
  
delete p7; .....
```



# Esercizio 3: Soluzione

```
1 // B() C() D() E() F()
2 // C() D() E() Fc
3 // B::f E::g E::j
4 // B::m E::g E::j
5 //NON COMPILA "error: passing 'const B' as 'this' argument discards qualifiers"
6 //(d//RUNTIME ERROR
7 // B::f C::g B::j
8 // B::m C::g
9 // B::j C::g
10 // B::f B::g D::j
11 // B::k D::j B::m B::g D::j
12 // B::n B::m B::g D::j
13 // B::n D::g
14 // B::f E::g E::j
15 // C::k B::n
16 // E::n B::m E::g E::j
17 //NON COMPILA "error: passing 'const B' as 'this' argument discards qualifiers"
18 //NON COMPILA "cannot dynamic_cast 'p6' (of type 'const class B*') to type 'class E*'"
19 // F::k E::g
20 // ~F() ~E() ~D() ~C() ~B()
```