Tutorato 9

17/01/2024

Programmazione ad Oggetti – 2023-2024

Gabriel Rovesti

2103389 – LM Computer Science

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO MATEMATICA

First thing first...

Registrazione presenza Tutorato PaO

Login con SSO su Google Form inserendo i propri dati

Casi particolari dei Cosa Stampa


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

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

```
class C: public A {
private:
  void j() { cout <<" C::j "; }
public:
  virtual void g() { cout <<" C::g "; }
  void m() { cout <<" C::m "; g(); j(); }
  void k() const { cout <<" C::k "; k(); }
};
```

```
class D: public B {
protected:
  void j() { cout <<" D::j "; }
public:
  B* n() final { cout <<" D::n "; return this; }
  void m() { cout <<" D::m "; g(); j(); }
};
```

```
A* p1 = new D(); A* p2 = new B(); A* p3 = new C(); B* p4 = new D(); const A* p5 = new C();
```

```
(static_cast<B*>(p3->n()))->g();
```

Casi particolari dei Cosa Stampa

Stampe → `A::n A::g`

Spiegazione

La conversione viene effettuata sullo *this.

Questa fa in modo che quando da A andiamo verso C, stiamo cercando di convertire a B. Lo *this riparte sempre dalla classe base. Nel momento in cui cerchiamo di andare verso una sottoclasse allo stesso livello, per essere «safe» rimane in A.

Morale

Tutte le stampe su *this restano nella classe base.

```
(static_cast<B*>(p3->n()))->g();
```

Casi particolari dei Cosa Stampa


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

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

```
class C: public A {
private:
  void j() { cout <<" C::j "; }
public:
  virtual void g() { cout <<" C::g "; }
  void m() { cout <<" C::m "; g(); j(); }
  void k() const { cout <<" C::k "; k(); }
};
```

```
class D: public B {
protected:
  void j() { cout <<" D::j "; }
public:
  B* n() final { cout <<" D::n "; return this; }
  void m() { cout <<" D::m "; g(); j(); }
};
```

```
A* p1 = new D(); A* p2 = new B(); A* p3 = new C(); B* p4 = new D(); const A* p5 = new C();
```

```
(p3->n())->m();
```

Casi particolari dei Cosa Stampa

Stampe → `A::n  A::m  A::g  C::j`

Spiegazione

La conversione viene effettuata sullo *this.

Quando ritorniamo dallo *this, «ripartiamo dalla classe base»; questo fa in modo che le successive stampe partano da A e poi se sono ridefinite vanno verso i sottotipi.
Il metodo m() non ha virtual ed essendo che ripartiamo da A stampiamo quello

Morale

Se ritorniamo con lo *this e il metodo non è virtual, stampiamo quello della classe base.

```
(p3->n())->m();
```

Casi particolari dei Cosa Stampa


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

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

class C: public A {
private:
  void j() { cout <<" C::j "; }
public:
  virtual void g() { cout <<" C::g "; }
  void m() { cout <<" C::m "; g(); j(); }
  void k() const { cout <<" C::k "; k(); }
};

class D: public B {
protected:
  void j() { cout <<" D::j "; }
public:
  B* n() final { cout <<" D::n "; return this; }
  void m() { cout <<" D::m "; g(); j(); }
};

A* p1 = new D(); A* p2 = new B(); A* p3 = new C(); B* p4 = new D(); const A* p5 = new C();
```

```
(static_cast<C*>(p2))->k();
```

Casi particolari dei Cosa Stampa

Stampe → `C::k (infinitamente... stack overflow!)`

Spiegazione

La conversione non viene effettuata sullo *this e quindi va bene.

Il metodo k() è presente ricorsivamente e viene chiamato indefinitamente; se non si ha il controllo di quello che fa il programma, si ha «undefined behaviour».
Correttamente, questa stampa, se non è presente l'opzione UB, si mette errore run-time.

Morale

Occhio se il metodo viene chiamato ricorsivamente o meno.

```
(static_cast<C*>(p2))->k();
```

Casi particolari dei Cosa Stampa

UNIVERSITÀ DEGLI STUDI DI PADOVA

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

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

class C: public A {
private:
  void j() { cout <<" C::j "; }
public:
  virtual void g() { cout <<" C::g "; }
  void m() { cout <<" C::m "; g(); j(); }
  void k() const { cout <<" C::k "; k(); }
};

class D: public B {
protected:
  void j() { cout <<" D::j "; }
public:
  B* n() final { cout <<" D::n "; return this; }
  void m() { cout <<" D::m "; g(); j(); }
};

A* p1 = new D(); A* p2 = new B(); A* p3 = new C(); B* p4 = new D(); const A* p5 = new C();
```

```
(p5->n())->g();
```

Casi particolari dei Cosa Stampa

Stampe → `NC (Non compila)`

Spiegazione

Banalmente, il puntatore è `const` e ritorno il `this` costante su un metodo non marcato costante. Quindi, se provo a tornarlo, dà errore.

Morale

Se c'è il `const` e ritorno `*this` dà errore

```
(p5->n())->g();
```

Casi particolari dei Cosa Stampa

Lo speculare del precedente è una cosa del tipo:

```
1  B* p1 = new E();
2
3  virtual const B* j() {return *this};
4
5  (p1->j())->k();
```

- Se ho classe * e ho il metodo `const Classe *` (che ha il `const`) e ritorna `this` è errore
- Stampe → `NC (Non compila)`

Casi particolari dei Cosa Stampa


```
class Z {
 public: Z(int x) {}
};
```

```
class A {
 public:
  void f(int) {cout << "A::f(int) "; f(true);}
  virtual void f(bool) {cout << "A::f(bool) ";}
  virtual A* f(Z) {cout << "A::f(Z) "; f(2); return this;}
  A() {cout << "A() "; }
};
```

```
class B: virtual public A {
 public:
  void f(const bool&) {cout << "B::f(const bool&) ";}
  void f(const int&) {cout << "B::f(const int&) ";}
  virtual B* f(Z) {cout << "B::f(Z) "; return this;}
  virtual ~B() {cout << "~B() ";}
  B() {cout << "B() "; }
};
```

```
class C: virtual public A {
 public:
  C* f(Z) {cout << "C::f(Z) "; return this;}
  C() {cout << "C() "; }
};
```

```
class D: virtual public A {
public:
  virtual void f(bool) const {cout << "D::f(bool) ";}
  A* f(Z) {cout << "D::f(Z) "; return this;}
  ~D() {cout << "~D() ";}
  D() {cout << "D() ";}
};
```

```
class E: public C {
 public:
  C* f(Z) {cout << "E::f(Z) "; return this;}
  ~E() {cout << "~E() ";}
  E() {cout << "E() ";}
};
```

```
class F: public B, public E, public D  {
 public:
  void f(bool) {cout << "F::f(bool) ";}
  F* f(Z) {cout << "F::f(Z) "; return this;}
  F() {cout << "F() "; }
  ~F() {cout << "~F() ";}
};
```

```
B *pb1 = new F; B *pb2 = new B;
C *pc = new C; E *pe = new E;
A *pa1 = pc, *pa2 = pe, *pa3 = new F;
```

```
(dynamic_cast<C*>(pa3))->f(Z(2));
```

Casi particolari dei Cosa Stampa

Stampe → `F::f(Z)`

Spiegazione

Sopra c'è l'oggetto di tipo C, in F si ha l'oggetto di tipo F* ed essendo fatta la chiamata direttamente sul puntatore, prende il tipo «più vicino» a quello dell'oggetto puntato.

Morale

Attenzione a considerare le ridefinizioni dei metodi; questo è un caso particolare, una sorta di «overriding implicito»

```
(dynamic_cast<C*>(pa3))->f(Z(2));
```

Cosa Stampa: Costruzioni e distruzioni

- Costruzione
  - Seguiamo l'ordine di ereditarietà
  - In caso di ereditarietà multipla, parto a costruire da sinistra
  - Tendenzialmente costruendo usando il tipo dinamico

- `E.g. B* p1 = new F(); //B() C() D() E() F()`
- `E.g. B* p1 = pf; //B() C() D() E() F()`
  - Altrimenti, uso il tipo statico

- `E.g. B* b; //B()`

- Occhio sempre a controllare che i costruttori siano tracciati!

Cosa Stampa: Costruzioni e distruzioni

- Distruzione
  - Seguiamo l'ordine di ereditarietà
  - In caso di ereditarietà multipla, parto a distruggere da destra
  - Tendenzialmente distruggo partendo dal tipo dinamico

- `E.g. B* p1 = new F(); //~F() ~E() ~D() ~C() ~B()`
- `E.g. B* p1 = pf; //~F() ~E() ~D() ~C() ~B()`
  - Altrimenti, uso il tipo statico

- `E.g. B* b; //~B()`

- Occhio sempre a controllare che i distruttori siano tracciati!

Cosa Stampa: track da seguire

1. Guardiamo il tipo statico e tipo dinamico

2. Se c'è il virtual, vado nella sottoclasse (tipo dinamico)

3. Se non c'è nulla nella sottoclasse, guardo se c'è un metodo con un tipo di ritorno «più vicino possibile» a quello che abbiamo
    1. E.g. se non c'è int, spesso, usiamo Z

4. Altrimenti, vado in cerca nelle altre classi della gerarchia
    1. E.g. se TS è B e tipo dinamico è D, se B è virtual e in D non c'è nulla, vado in C

static_cast e dynamic_cast

Come funzionano lo `static_cast` ed il `dynamic_cast`?

- Il primo cerca di convertire staticamente l'oggetto (quindi, prima che venga eseguito) oppure prende l'oggetto puntato e cerca subito di convertirlo
  - Se non ci sono tipi compatibili, o rimane nella classe base o dà errore

- Il secondo cerca di convertire l'oggetto a runtime (quindi, durante l'esecuzione) e può essere più imprevedibile
  - Se non ci sono tipi compatibili, dà errore o cerca il metodo «più vicino» nella sua gerarchia

- In entrambi i casi, quello che cambia è il tipo statico (se non viene fatto dopo lo *this, di solito non dà problemi)

E.g. → `A* p1 = new D();` e `(dynamic_cast<B*>(p1))->m();`
- TS – diventa B TD – resta D

E.g. → `A* p1 = new D();` e `(static_cast<B*>(p1))->m();`
- TS – diventa B TD – resta D

Funzioni: track da seguire

- Capisco quali metodi mi offre la gerarchia
  - Se si tratta di una modellazione, la gerarchia devo farla «a mano», poi il secondo pezzo mi chiede di definire la gerarchia della classe con vector

- Comincio a scrivere la funzione richiesta capendo a che tipi convertire

- Generalmente, si ha una struttura con un ciclo e operazioni su `list` e `vector`
  - Se l'oggetto puntato non è `const`, si usa il tipo `vector` e `iterator`
  - Se l'oggetto puntato è `const`, si usa il tipo `vector` e `const_iterator`
- In entrambi i casi, si può usare `auto` senza problemi
  - Se volete usare il `for` range-based, penso non ci siano problemi

- In tutti i casi
  - Se è necessario controllare «se il tipo dinamico è esattamente uguale ad un altro tipo dinamico», uso `typeid`
  - Se è necessario controllare «se il tipo dinamico è il sottotipo proprio di un'altra classe», uso `dynamic_cast`

Modellazioni: track da seguire

- Capisco quali metodi mi offre la gerarchia
  - Se si tratta di una modellazione, la gerarchia devo farla «a mano», poi il secondo pezzo mi chiede di definire la gerarchia della classe con `vector`

- Comincio a scrivere le classi, ricordandomi di:
  - Inserire i costruttori
  - Inserire i metodi `get` per i campi privati
  - Inserire il distruttore virtuale nella classe base della gerarchia per renderla polimorfa
  - Se ho dei metodi virtuali puri, ogni classe, se specificato, li deve ridefinire

- Nel secondo pezzo dell'esercizio
  - Definisco le strutture dati da usare (di solito, per i motivi visti nel corso), uso vector

- Seguo le stesse regole delle funzioni (dal punto 2 al punto 6)

Definizioni standard: track da seguire

- Capisco da quali classe deriva la mia (guardo solo le sottoclassi dirette)
  - Per ereditarietà, assumiamo che le sottoclassi siano già state costruite con le superclassi non dirette (funziona così)
- Definisco la firma dell'operatore di assegnazione/costruttore di copia
- Richiamo l'operatore di assegnazione (con scoping) o il costruttore di copia (semplicemente usando classe(parametro)
- Assegno gli altri parametri
- Attenzione: nel costruttore di copia non è necessario fare il controllo su *this != puntatore; non è il goal di questo tipo di esercizio (e neanche le soluzioni ufficiali lo prevedono)

Gerarchie: track da seguire

- Seguo letteralmente cosa mi chiedono i metodi, ricordando che:
  - Se una classe deve essere polimorfa, metto il distruttore virtuale
  - Se una classe deve essere astratta, metto il metodo virtuale puro
    - `E.g. virtual void method()=0`
  - Se una classe deve essere concreta, ridefinisco il metodo virtuale puro
    - `E.g. virtual void method()`
  - Se una classe dice di «non permettere la costruzione pubblica dei suo oggetti, ma solamente la costruzioni di oggetti D che siano sottooggetti», uso protected
  - Gli altri punti riguardano le assegnazioni seguendo il tipo standard (e.g. regole definite nella slide «Definizioni standard»

Sottotipi/Stampe con numeri: track da seguire

- Parto dalle cose certe (sapendo che stampo la stringa «X» ed «Y», seguo letteralmente quello che dice il metodo e capisco un po' di informazioni dalle chiamate
- Date le certezze, cerco poi di capire cosa altro stampare
- Le altre righe del main compilano (lo specifica l'esercizio) e si assume che le conversioni vadano bene
- Tendenzialmente
  - si capisce subito che se un tipo dinamico è assegnato ad un tipo statico, è un suo sottotipo
  - controllo se le conversioni dei `dynamic_cast` vanno bene o meno
    - ricordiamo che la struttura è `dynamic_cast<sottotipo*>(supertipo)`

Esercizio 1: Sottotipi

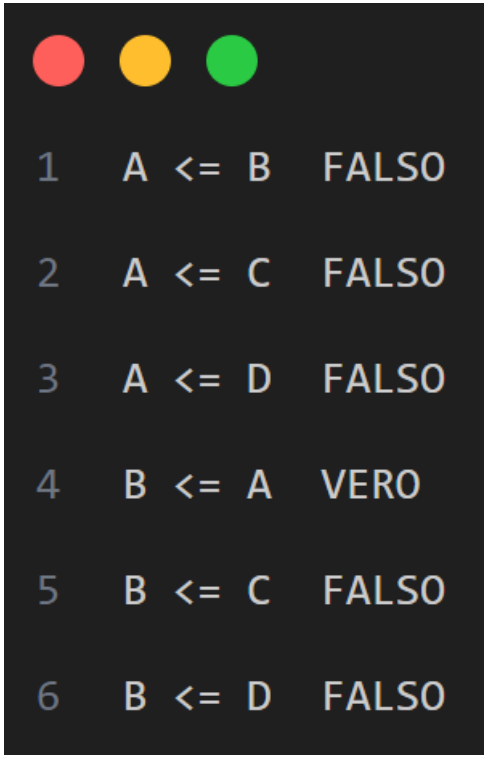
quesito3.cpp 1

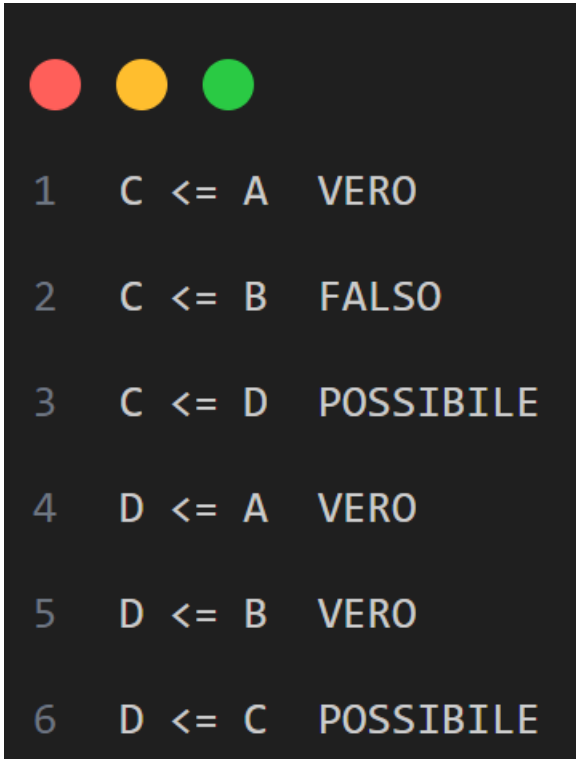
```
template <class X,class Y>

X*Fun(X*p){return dynamic_cast<Y*>(p);}

main(){
    C c; fun<A,B>(&c);
    if(fun<A,B>(new C()==0)) cout<<"Alan";
    if(dynamic_cast<C*>(new B())==0) cout<<"Turing";
    A*p=fun<D,B>(new D());
}
```

	Vero	Falso	Possibile
A<=B			
A<=C			
A<=D			
B<=A			
B<=C			
B<=D			

	Vero	Falso	Possibile
C<=A			
C<=B			
C<=D			
D<=A			
D<=B			
D<=C			

Esercizio 1: Soluzione





Esercizio 2: Cosa Stampa

Esercizio Cosa Stampa

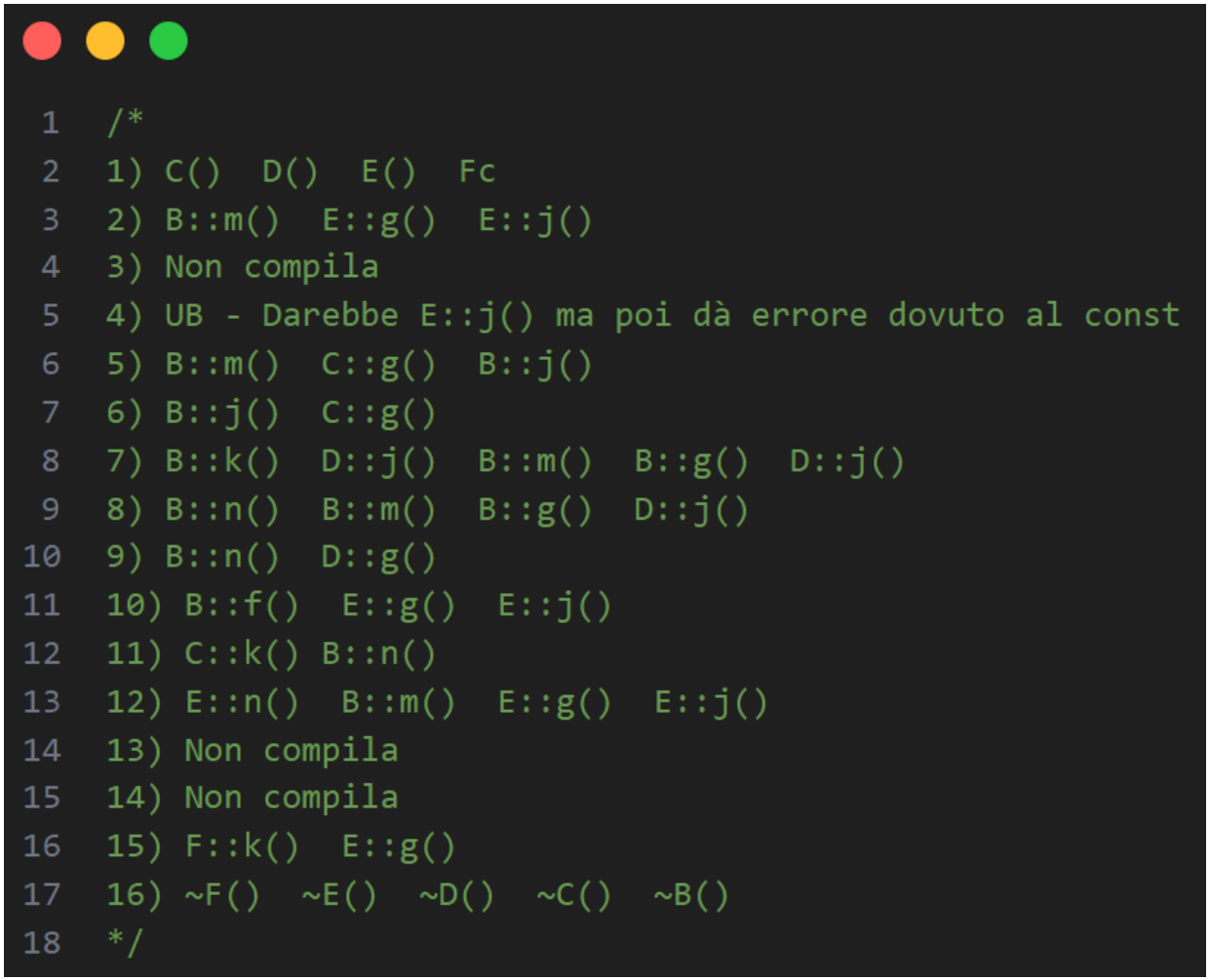
```
class B {
public:
  B() {cout<< " B() ";}
  virtual ~B() {cout<< " ~B() ";}
  virtual void f() {cout <<" B::f "; g(); j();}
  virtual void g() const {cout <<" B::g ";}
  virtual const B* j() {cout<<" B::j "; return this;}
  virtual void k() {cout <<" B::k "; j(); m(); }
  void m() {cout <<" B::m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

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();}
};

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;
```

Esercizio 2: Cosa Stampa

```
C* ptr = new F(f); ......................
p1->m(); ...............................
(p1->j())->k(); ........................
(dynamic_cast<const F*>(p1->j()))->g(); .
p2->m(); ...............................
(p2->j())->g(); ........................
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 2: Soluzione



```
/*
1) C()  D()  E()  Fc
2) B::m()  E::g()  E::j()
3) Non compila
4) UB - Darebbe E::j() ma poi dà errore dovuto al const
5) B::m()  C::g()  B::j()
6) B::j()  C::g()
7) B::k()  D::j()  B::m()  B::g()  D::j()
8) B::n()  B::m()  B::g()  D::j()
9) B::n()  D::g()
10) B::f()  E::g()  E::j()
11) C::k() B::n()
12) E::n()  B::m()  E::g()  E::j()
13) Non compila
14) Non compila
15) F::k()  E::g()
16) ~F()  ~E()  ~D()  ~C()  ~B()
*/
```

Esercizio 3: Stampa numerica

Esercizio Tipi

```
class A {                                   class C: virtual public B {};
public:
  virtual ~A() = 0;                         class D: virtual public B {};
};
A::~A() = default;                          class E: public C, public D {};

class B: public A {
public:
  ~B() = default;
};

char F(const A& x, B* y) {
  B* p = const_cast<B*>(dynamic_cast<const B*> (&x));
  auto q = dynamic_cast<const C*> (&x);
  if(dynamic_cast<E*> (y)) {
    if(!p || q) return '1';
    else return '2';
  }
  if(dynamic_cast<C*> (y)) return '3';
  if(q) return '4';
  if(p && typeid(*p) != typeid(D)) return '5';
  return '6';
}
```

```
int main() {
B b; C c; D d; E e;


cout << F(...,...) << F(...,...) << F(...,...) << F(...,...) << F(...,...)


     << F(...,...) << F(...,...) << F(...,...) << F(...,...) << F(...,...);
}
```

Si considerino le precedenti definizioni ed il `main()` incompleto. Definire opportunamente negli appositi spazi `...,...` le chiamate alla funzione `F` di questo `main()` usando gli oggetti locali `b`, `c`, `d`, `e`, `f` in modo tale che: (1) non vi siano errori in compilazione o a run-time; (2) le chiamate di `F` siano **tutte diverse** tra loro; (3) l'esecuzione produca in output **esattamente** la stampa **`6544233241`**.

Esercizio 3: Soluzione (possibile)

```
1  int main()
2  {
3      std::cout<<fun(d,&d)<<fun(b,&d)<<fun(c,&d)<<fun(e,&d)
4      <<fun(d,&e)<<fun(c,&c)<<fun(d,&c)<<fun(b,&e)<<fun(c,&b)<<fun(c,&e);
5  }
```

Esercizio 4: Ridefinizione standard

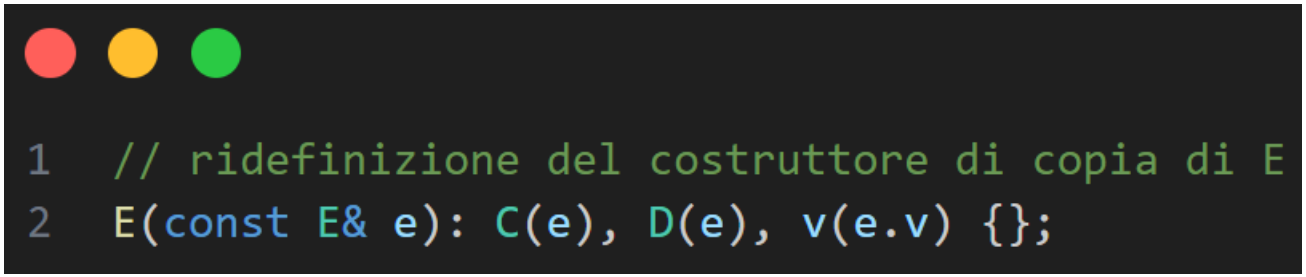
Si considerino le seguenti definizioni.

```
class B {
private:
  list<double>* ptr;
  virtual void m() =0;
};

class D: virtual public B {
private:
  int x;
};

class E: public C, public D {
private:
  vector<int*> v;
public:
  void m() {}
  // ridefinizione del costruttore di copia di E
};
```

```
class C: virtual public B {};
```

Ridefinire il costruttore di copia di `E` in modo tale che il suo comportamento coincida con quello del costruttore di copia standard di `E`.

Esercizio 4: Soluzione



```
1  // ridefinizione del costruttore di copia di E
2  E(const E& e): C(e), D(e), v(e.v) {};
```