(P4 ->N()).M();

Esercizio Cosa Stampa

```
class A {
public:
  A() {cout<< " A() ";}
  ~A() {cout<< " ~A ";}
  A(const A& x) {cout<< " Ac ";}
  virtual const A* j() {cout<<" A::j "; return this;}
  virtual void k() {cout <<" A::k "; m();}
  void m() {cout <<" A::m "; j();}
};

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

class C: virtual public B {
public:
  C() {cout<< " C() ";}
  ~C() {cout<< " ~C ";}
  void g() const {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 ";}
  E(const E& x) {cout<< " Ec ";}
  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;}
};

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

VIRTUAL PUBLIC

E: A() B() C() D() E()

FINAL → NO OVERRIDING ULTERIORE (NO MORE GOING DOWN)

return *this = Ritorna alla classe base (più base possibile) data la costruzione della gerarchia

In particolare, "virtual public" assicura una sola costruzione del sottooggetto.

(dynamic_cast<D*>(p4))->n()).k();

Esercizio Cosa Stampa

```
class A {
public:
  A() {cout<< " A() ";}
  ~A() {cout<< " ~A ";}
  A(const A& x) {cout<< " Ac ";}
  virtual const A* j() {cout<<" A::j "; return this;}
  virtual void k() {cout <<" A::k "; m();}
  void m() {cout <<" A::m "; j();}
};

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

class C: virtual public B {
public:
  C() {cout<< " C() ";}
  ~C() {cout<< " ~C ";}
  void g() const {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 ";}
  E(const E& x) {cout<< " Ec ";}
  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;}
};

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

B * P4 = NOW E();

D * P4 = NOW E();

Esercizio Tipi. Siano `A`, `B`, `C`, `D` e `E` cinque **diverse** classi polimorfe. Si considerino le seguenti definizioni.

```
template <class X, class Y>
const X& fun(const Y& r) {
  const Y* ptr = &r; return dynamic_cast<const X*>(ptr);
}
int main() {
  B b; C c; D d; E e;
  if( fun<A,B>(c) == nullptr ) cout << "Bjarne ";
  if( dynamic_cast<C*>(&b) == nullptr ) cout << "Stroustrup";
  const A* p = fun<D,B>(d);
  const D* q = fun<E,B>(e);
  fun<C,D>(d);
}
```

Si supponga che:

1. il `main()` compili correttamente ed esegua senza provocare errori a run-time o undefined behaviour;
2. l'esecuzione del `main()` provochi in output su `cout` la stampa `Bjarne Stroustrup`.

In tali ipotesi, per ognuna delle relazioni di sottotipo `T1`≤`T2` nella seguente **tabella da ricopiare nel foglio** scrivere nella corrispondente cella:

(a) "VERO" per indicare che `T1` **sicuramente è** sottotipo di `T2`;

(b) "FALSO" per indicare che `T1` **sicuramente non è** sottotipo di `T2`;

(c) "POSSIBILE" **altrimenti**, ovvero se non valgono nè (a) nè (b).

A≤B	A≤C	A≤D	A≤E	B≤E	D≤E
F	F	F	F	F	F