

$(p4 \rightarrow N()) \cdot M()$

VIRTUAL PUBLIC

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 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 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();
```

```
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 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();}
};
```

B: A() B()

C()

D()

E()

FINAL → NO OVERRIDING
ULTIMATES
(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.

$(dynamic_cast<D*>(p4)) \rightarrow n()) \cdot k();$

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 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 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();
```

```
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 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();}
};
```



$B * p4 = \text{new } B();$

$D * p4 = \text{new } B();$

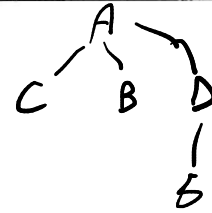
$B \rightarrow \text{CONST } A \dots \rightarrow B \leq A$ (sottotipo vero)

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

```
template <class B, class Y>
const B fun(B r) {
    const B ptr = &r; return dynamic_cast<B*>(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);
}
```

$B \rightarrow \text{CONST } A \dots \rightarrow B \leq A$ (?)
 \downarrow BAD-CAST (STD::) / 10



$\rightarrow C \not\leq B$

$\rightarrow B \not\leq C$

$A \times A = \text{NON D.C. TS/TO}$

Si supponga che:

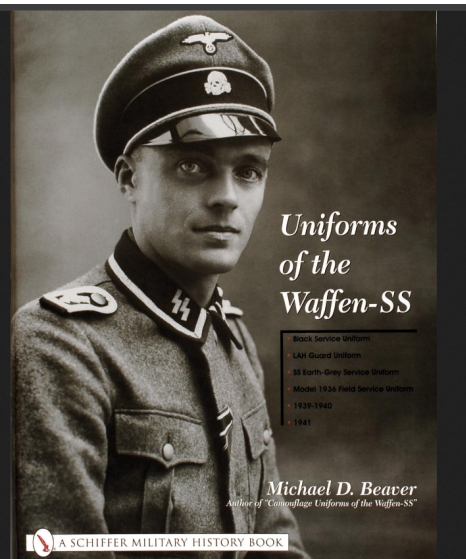
- il main() compili correttamente ed esegua senza provocare errori a run-time o undefined behaviour;
- l'esecuzione del main() provochi in output su cout la stampa Bjarne Stroustrup.

In tali ipotesi, per ognuna delle relazioni di sottotipo $T1 \leq T2$ nella seguente tabella da ricopiare nel foglio scrivere nella corrispondente cella:

- "VERO" per indicare che $T1$ sicuramente è sottotipo di $T2$;
- "FALSO" per indicare che $T1$ sicuramente non è sottotipo di $T2$;
- "POSSIBILE" altrimenti, ovvero se non valgono nè (a) nè (b).

$A \leq B$	$A \leq C$	$A \leq D$	$A \leq E$	$B \leq E$	$D \leq E$
F	F	F	F	F	F

NO COMPILAZIONE
SUI CAST...



RANGABEN \rightarrow SA!