

```

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

class B: virtual public A {};

class C: virtual public A {
public:
    virtual void m() {}
};

class D: public B, public C {
public:
    virtual void m() {}
};

class E: public D {};

class F: public E {};

```

[S]

F(B, F)

```

char G(A* p, B& r) {
    C* pc = dynamic_cast<E*>(&r);
    if(pc && typeid(*p)==typeid(r)) return 'G';
    if(!dynamic_cast<E*>(&r) && dynamic_cast<D*>(p)) return 'Z';
    if(!dynamic_cast<F*>(pc)) return 'A';
    else if(typeid(*p)==typeid(E)) return 'S';
    return 'E';
}

```

usato
PZZ

!E
!S non E = Z !E
P != E → P == D!

```

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

class B: virtual public A {};

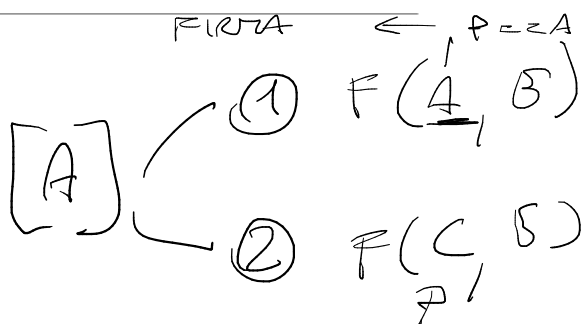
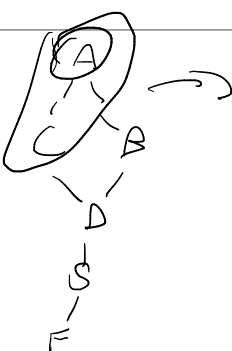
class C: virtual public A {
public:
    virtual void m() {}
};

class D: public B, public C {
public:
    virtual void m() {}
};

class E: public D {};

class F: public E {};

```



SOMMARIO
DIRUTTO

```

char G(A* p, B& r) {
    C* pc = dynamic_cast<E*>(&r);
    if(pc && typeid(*p)==typeid(r)) return 'G';
    if(!dynamic_cast<E*>(&r) && dynamic_cast<D*>(p)) return 'Z';
    if(!dynamic_cast<F*>(pc)) return 'A';
    else if(typeid(*p)==typeid(E)) return 'S';
    return 'E';
}

```

P != E
PC
R != D (A, S)
(C, B)

Si consideri inoltre il seguente statement.

```

cout << G(new X1,*new Y1) << G(new X2,*new Y2) << G(new X3,*new Y3) << G(new X4,*new Y4)
    << G(new X5,*new Y5) << G(new X6,*new Y6) << G(new X7,*new Y7) << G(new X8,*new Y8);

```



CASO IN CUI PC → E ...

SO NON VIENE CONSIDERATO [VARIE
OPZIONI!
!!]

```

char G(A* p, B& r) {
    C* pc = dynamic_cast<E*>(&r);
    if(pc && typeid(*p)==typeid(r)) return 'G';
    if(!dynamic_cast<E*>(&r) && dynamic_cast<D*>(p)) return 'Z';
    if(!dynamic_cast<F*>(pc)) return 'A';
    else if(typeid(*p)==typeid(E)) return 'S';
    return 'E';
}

```

F(C, D) / F(B, B)

```

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

class B: virtual public A {};

class C: virtual public A {
public:
    virtual void m() {}
};

class D: public B, public C {
public:
    virtual void m() {}
};

class E: public D {};

class F: public E {};

```

① F(B,E)
② F(E,E)

↓
SOSTITUIRE
DIRETTO
DLB

```

char G(A* p, B& r) {
    C* pc = dynamic_cast<E*>(&r);
    if(pc && typeid(*p)==typeid(r)) return 'G';
    if(!dynamic_cast<E*>(&r) && dynamic_cast<D*>(p)) return 'Z';
    if(!dynamic_cast<F*>(pc)) return 'A';
    else if(typeid(*p)==typeid(E)) return 'S';
    return 'E';
}

```

→ E==5 (5,6)

```

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

class B: virtual public A {};

class C: virtual public A {
public:
    virtual void m() {}
};

class D: public B, public C {
public:
    virtual void m() {}
};

class E: public D {};

class F: public E {};

char G(A* p, B& r) {
    C* pc = dynamic_cast<E*>(&r);
    if(pc && typeid(*p)==typeid(r)) return 'G';
    if(!dynamic_cast<E*>(&r) && dynamic_cast<D*>(p)) return 'Z';
    if(!dynamic_cast<F*>(pc)) return 'A';
    else if(typeid(*p)==typeid(E)) return 'S';
    return 'E';
}

```

↓
B → (D,D)
D → (E,D)

① F(D,B)
② F(E,B)

P!=20, P!=B

P==E

→ P!=5, P=2D

SOSTITUIRE

```

char G(A* p, B& r) {
    C* pc = dynamic_cast<E*>(&r);
    if(pc && typeid(*p)==typeid(r)) return 'G';
    if(!dynamic_cast<E*>(&r) && dynamic_cast<D*>(p)) return 'Z';
    if(!dynamic_cast<F*>(pc)) return 'A';
    else if(typeid(*p)==typeid(E)) return 'S';
    return 'E';
}

```

[E]

= TUTTO FALLISCE

= COPIA IN P) DALLA FIRMA

→ F(A,B) / F(C,F)

Definire un template di funzione

```
template <class T> list<const istream*> compare(vector<ostream*>&, vector<const T*>&)
con il seguente comportamento: in ogni invocazione compare(v,w),
```

1. se v e w non contengono lo stesso numero di elementi allora viene sollevata una eccezione di tipo `string` che rappresenta la stringa vuota;
2. se v e w contengono lo stesso numero di elementi allora per ogni posizione i dentro i bounds dei due vettori v e w :
 - (a) se $*v[i]$ è un `fstream` ed è dello stesso tipo di $*w[i]$ allora: (i) il puntatore $v[i]$ viene inserito nella lista che la funzione deve ritornare; (ii) i puntatori $v[i]$ e $w[i]$ vengono rimossi dai vettori che li contengono;
 - (b) se $*w[i]$ è uno `stringstream` in stato `good` e $*v[i]$ e $*w[i]$ sono di tipo diverso allora il puntatore $w[i]$ viene inserito nella lista che la funzione deve ritornare.

```
template <class T>
```

```
std::list <const istream*> compare(std::vector<ostream*>& v, vector<const T*>& w){
```

```
    std::list <const istream*> l;
```

```
    // (a.1)
```

```
    if(v.size() != w.size()) throw std::string("");
```

```
    // (a.2) → Si noti che "v" e "w" hanno stessa size; basta solo "i"
```

```
    for(int i = 0, j = 0; i < v.size() && w.size(); i++, j++){ // con due (così vedi)
```

```
        ostream* f = dynamic_cast<ostream*>(*v[i]);
```

```
        if(f && typeid(f) == typeid(*w[j])){
```

```
            l.push_back(f);
```

```
            // Creo copie → sempre deallocare (C++ - no garbage collection)
```

```
            ostream* o = v[i]; → non const
```

```
            T* t = const_cast<T*>(*w[j]); → const (OCCHIO) → no const_erase
```

```
            // Caso particolare erase (normalmente porto alla posizione dopo)
```

```
            // ma qui la rottura è avere v[i] e w[j];
```

```
            // altrimenti, normalmente v.erase(it) oppure it = v.erase(it);
```

```
            v[i] = v.erase(o); // porta alla posizione dopo
```

```
            w[j] = w.erase(t);
```

```
            // Deallocare le copie
```

```
            delete(o);
```

```
            delete(t);
```

```
        }
```

```
        // (b)
```

```
        stringstream* s = dynamic_cast<stringstream*>(*w[j]);
```

```
        if(s && s->good() && typeid(s) != typeid(*v[i])){
```

```
            l.push_back(s);
```

```
        }
```

```
    }
```

```
    return l;
```

```
}
```

Siano A, B, C e D quattro diverse classi polimorfe. Si considerino le seguenti definizioni.

```
template <class B> class B {
    fun(B p) { return dynamic_cast<B*>(p); }
};

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

Si supponga che:


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

In tali ipotesi, per ognuna delle relazioni di sottotipo $T_1 \leq T_2$ nelle seguenti tabelle segnare con una croce l'entrata

- (a) "Vero" per indicare che T_1 sicuramente è sottotipo di T_2 ;
- (b) "Falso" per indicare che T_1 sicuramente non è sottotipo di T_2 ;
- (c) "Possibile" altrimenti, ovvero se non valgono né (a) né (b).

	Vero	Falso	Possibile
$A \leq B$		X	
$A \leq C$		X	
$A \leq D$		X	
$B \leq A$	X		
$B \leq C$		X	
$B \leq D$			X

	Vero	Falso	Possibile
$C \leq A$	X		
$C \leq B$		X	
$C \leq D$		X	
$D \leq A$	X		
$D \leq B$			X
$D \leq C$		X	



Esercizio Gerarchia

Definire una **unica gerarchia di classi** che includa:

- (1) una **unica classe base polimorfa A** alla radice della gerarchia;
- (2) una **classe derivata astratta E**;
- (3) una **sottoclasse C di B** che sia **concreta**;
- (4) una **classe D** che non permetta la costruzione pubblica dei suoi oggetti, ma solamente la costruzione di oggetti di D che siano sottooggetti;
- (5) una **classe E** definita mediante **derivazione multipla a diamante con base virtuale**, che abbia D come supertipo, e con l'**assegnazione ridefinita pubblicamente con comportamento identico a quello dell'assegnazione standard** di E.

POLIMORFA → DISTRIBUZIONE VIRTUALE

VIRTUALE PURO

ASTRATTA → VIRTUAL INT SOMMA() = 0;

CONCRETO → VIRTUAL INT SOMMA().

↑
SUSPENDING

PUBLIC → PROTECTED

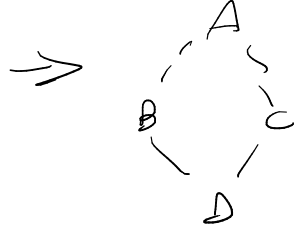
```
CLASS D {
    PRIVATE: int x;
    PROTECTED:
    D(x1(x)) &?;
```

EREDITARIETÀ
MULTIPLA



→ DIAMANTE...

VIRTUAL
PUBLIC



D: VIRTUALPUBLIC B;
VIRTUALPUBLIC C

D: PUBLIC B, PUBLIC C {

// ASSEGNAZIONE STANDARD

D & OPERATOR = (CONST D & D) {

B:: OPERATOR = (D)

C:: OPERATOR = (D);

↓
RETURN x & DLS;

ESERCIZIO PRO
GERARCHIA!

Ognuno dei seguenti frammenti è il codice di uno o più metodi pubblici di una qualche classe C. La loro compilazione provoca errori?

<code>C f(C& x) {return x;}</code>
<code>C& g() const {return *this;}</code>
<code>C h() const {return *this;}</code>
<code>C* m() {return this;}</code>
<code>C* n() const {return this;}</code>
<code>void p() {}</code> <code>void q() const {p();}</code>
<code>void p() {}</code> <code>static void r(C *const x) {x->p();}</code>
<code>void s(C *const x) const {*this = *x;}</code>
<code>static C& t() {return C();}</code>
<code>static C *const u(C& x) {return &x;}</code>

OK/NC?	→ RIFERIMENTO = OK
OK/NC?	→ CONST CORRUPTNESS
OK/NC?	→ NO RIF.
OK/NC?	→ GIUSTO
OK/NC?	→ CONST CORRUPTNESS
OK/NC?	.
OK/NC?	→ NON TOCCATO
OK/NC?	p() → WORKS IL CONST
OK/NC?	.
OK/NC?	.
OK/NC?	.

2. `void p() {} static void r(C *const x) {x->p();}` - Compila

- `r` è static e riceve un puntatore const a C (l'oggetto puntato è modificabile)
- La chiamata `x->p()` è valida poiché `p` è accessibile e il puntatore è dereferenziabile

3. `void s(C *const x) const {*this = *x;}` - Non compila

- `s` è const, quindi non può modificare i membri dell'oggetto
- L'assegnamento `*this = *x` tenta di modificare l'oggetto corrente, violando const

Il codice non compila per due ragioni sintattiche critiche:

1. Il tipo di ritorno `C&` richiede l'operatore `&` nella dichiarazione: `static C& t()`
2. Il return `C()` crea un oggetto temporaneo - non è possibile restituire un riferimento ad un oggetto temporaneo poiché questo verrebbe distrutto alla fine della funzione, lasciando un riferimento pendente

La correzione sarebbe: `static C t() {return C();}` per restituire l'oggetto per valore.

IT WORKS

