

```

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';
}

```

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);

```

Definire opportunamente le incognite di tipo X_i e Y_i tra i tipi A, B, C, D, E, F della precedente gerarchia in modo tale che:

1. Lo statement non includa più di una chiamata della funzione G con gli stessi parametri attuali
2. La compilazione dello statement non produca illegalità
3. L'esecuzione dello statement non provochi errori a run-time
4. L'esecuzione dello statement produca in output esattamente la stampa **SAGGEZZA**.

$TD(r) \in \{D, E, F\}$

$TD(*p) \in \{C, D, E, F\}$

output $G \in \{(E, E), (F, F)\}$

$TD(r) \leq E \ \& \ TD(*p) = TD(r)$

output $Z \in \{(C, D), (D, D)\}$

$\neg G \ \& \ TD(r) \not\leq E \ \& \ TD(*p) \leq D$

output $A \in \{(C, E), (D, E), (E, D), (F, E), (F, D)\}$

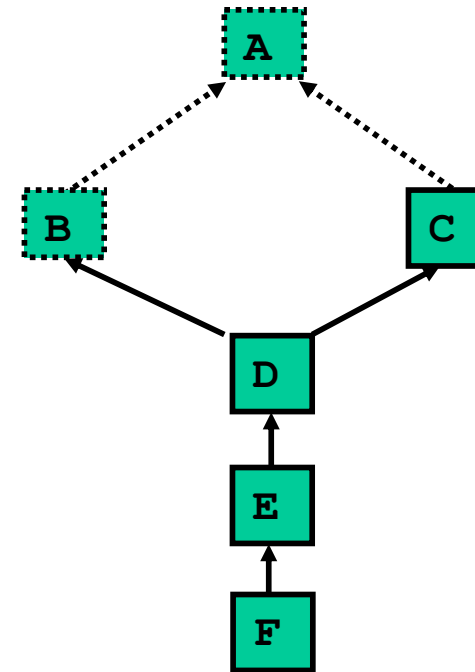
$\neg Z \ \& \ \neg G \ \& \ TD(r) \not\leq F$

output $S \in \{(E, F)\}$

$\neg Z \ \& \ \neg G \ \& \ \neg A \ \& \ TD(r) = F \ \& \ TD(*p) = E$

output $E \in \{(C, F), (D, F)\}$

$\neg Z \ \& \ \neg G \ \& \ \neg A \ \& \ \neg S$



```

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';
}

```

S=(E , F) A=(C , D) G=(E , E) G=(F , F)
E=(C , F) Z=(D , D) Z=(E , D) A=(C , E)

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);

```

Definire opportunamente le incognite di tipo X_i e Y_i tra i tipi A, B, C, D, E, F della precedente gerarchia in modo tale che:

1. Lo statement non includa più di una chiamata della funzione G con gli stessi parametri attuali
2. La compilazione dello statement non produca illegalità
3. L'esecuzione dello statement non provochi errori a run-time
4. L'esecuzione dello statement produca in output esattamente la stampa **SAGGEZZA**.

```

// dichiarazione incompleta
template<class T> class D;

template<class T1, class T2>
class C {
    // amicizia associata
    friend class D<T1>;
private:
    T1 t1; T2 t2;
};

template<class T>
class D {
public:
    void m() {C<T,T> c;
               cout << c.t1 << c.t2;}
    void n() {C<int,T> c;
               cout << c.t1 << c.t2;}
    void o() {C<T,int> c;
               cout << c.t1 << c.t2;}
    void p() {C<int,int> c;
               cout << c.t1 << c.t2;}
    void q() {C<int,double> c;
               cout << c.t1 << c.t2;}
    void r() {C<char,double> c;
               cout << c.t1 << c.t2;}
};

```



I seguenti main() compilano?

main()	{D<char> d; d.m();}	// C
main()	{D<char> d; d.n();}	// NC
main()	{D<char> d; d.o();}	// C
main()	{D<char> d; d.p();}	// NC
main()	{D<char> d; d.q();}	// NC
main()	{D<char> d; d.r();}	// C

```
class Z {
private:
    int x;
};

class B {
private:
    Z x;
};

class D: public B {
private:
    Z y;
public:
    // ridefinizione di operator=
    ...
};
```

Ridefinire l'assegnazione `operator=` della classe `D` in modo tale che il suo comportamento coincida con quello dell'assegnazione standard di `D`.

```
D& operator=(const D& d) {
    B::operator=(d);
    y=d.y;
    return *this;
}
```