```
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';
}
```

(SAGGEZZA)

→ S (x1) = G(B, F)

→ A (x 2) ← G (A, B) / G(C, D)

→ G(x 2) (G(B,B) G(F, B)

→ B (x 1) → G(A, B)

→ Z(x2) (G(D, B) G(D, D)

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 `Xi` e `Yi` tra i tipi `A`, `B`, `C`, `D`, `E`, `F` della precedente gerarchia in modo tale che:

1. Lo statement non includa piú 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**.

```
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';
}
```

R. != B

C ≤ F

P == S

P != D

(SAGGEZZA)

→ S (x1) = G(B, F)

→ A (x 2)

→ G (x 2)

→ B (x 1)

→ Z (x 2)

```
};

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';
}
```

R. != S

P. != D

R. != F

(SAGGEZZA)

→ S (x1) = G(B, F)

→ A (x 2) ← G (A, B) / G(C, D)

→ G (x 2)

→ B (x 1)

→ Z (x 2)

```
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';
}
```

(SAGGEZZA)

→ S (x1) = G(E, F)

→ A (x 2)

→ G (x 2)

→ E (x 1)

→ Z (x 2)

R != E

C ≤ F

P = 2E

P != 0

```
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';
}
```

E V

CONST E V

(L == E)

(SAGGEZZA)

→ S (x1) = G(E, F)

→ A (x 2) = G (A, B) / G(C, D)

→ G (x 2) (G(E,E) G(F, E)

→ E (x 1)

→ Z (x 2)

NB: Scrivere la soluzione **chiaramente** nel foglio a quadretti. Per comodità di correzione, **definire tutti i metodi inline**.

Esercizio 3 C++

Definire una unica gerarchia di classi che includa:

(1) una classe base polimorfa A alla radice della gerarchia;

(2) una classe derivata astratta B;

(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 derivata direttamente da D e con l'assegnazione ridefinita pubblicamente con comportamento identico a quello dell'assegnazione standard di E.

NB: Scrivere la soluzione **chiaramente** nel foglio a quadretti.

```
class A{
        virtual ~A();
};

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

class E: public D{
        // assegnazione standard
        E& operator=(const E& e){
                D::operator=(e);
                return *this;
        }
}
```

```
class C: public B{
        public:
                virtual void method() {
                        return;
                }
};

class D{
        private:
                int x;
        protected:
                D(int x1): x1(x);
}
```

Quesito 2

```
class A { public: virtual ~A() {} };

class B: virtual public A {};

class C: virtual public A {};

class D: public B, public C {};

class E: public D {};

class F: public E {};

template<class T>
void Fun(T& ref){
  bool b=0;
  B* p = &ref;
  try{ throw ref; }
  catch(E){cout << "E "; b=1;}
  catch(D){cout << "D "; b=1;}
  catch(B){cout << "B "; b=1;}
  catch(A){cout << "A "; b=1;}
  catch(C){cout << "C "; b=1;}
  if(b==0) cout << "ZERO ";
}

A a; B b; C c; D d; E e; F f;
A* pa = &b; D* pd = &f;
B* pb=dynamic_cast<B*>(pa); C* pc=dynamic_cast<E*>(pd); E* pe=static_cast<E*>(pd);
```

Fun(a);
Fun(b);
Fun(c);
Fun(d);
Fun(e);
Fun(f);
Fun(*pa);
Fun(*pb);
Fun(*pc);
Fun(*pd);
Fun(*pe);
Fun<B>(*pd);
Fun<D>(*pd);
Fun<E>(*pd);