Si assuma che `Abs` sia una classe base astratta fissata. Definire un template di funzione `bool Fun(T1*, T2&)`, dove `T1` e `T2` sono parametri di tipo, con il seguente comportamento. Si consideri una istanziazione implicita `Fun(ptr,ref)` dove si suppone che i parametri di tipo `T1` e `T2` siano istanziati a tipi polimorfi. Allora `Fun(ptr,ref)` ritorna `true` se e soltanto se valgono le seguenti due condizioni:

1. I parametri di tipo `T1` e `T2` sono istanziati allo stesso tipo;
2. Siano `D1*` il tipo dinamico di `ptr` e `D2&` il tipo dinamico di `ref`; allora: (i) `D1` e `D2` sono lo stesso tipo e (ii) questo tipo è un sottotipo proprio della classe `Abs`.

Scrivere la risposta nel riquadro sotto.

```cpp
#include <typeinfo> // usare typeid

template <class T1, class T2>
bool Fun(T1* ptr, T2& ref){
    // typeid = uguaglianza tra tipi polimorfi

    // T1/T2 sono istanziati allo stesso tipo
    if(typeid(*ptr) == typeid(ref)) // (1)

    if(typeid(T1) == typeid(T2) &&
    dynamic_cast<Abs*>(*ptr)) // (2)

    return true;

    // dynamic_cast = safe downcasting
    // X è un sottotipo proprio di Y
}
```

Tipi

Parametri

dynamic_cast<Sottotipo*>(*obj)

typeid(*obj) OR typeid(T1)

T1 ≤ C

Sia `B` una classe polimorfa e sia `C` una sottoclasse di `B`. Definire una funzione `int Fun(const vector<B*>& v)` con il seguente comportamento: sia `v` non vuoto e sia `T*` il tipo dinamico di `v[0]`; allora `Fun(v)` ritorna il numero di elementi di `v` che hanno un tipo dinamico `T1*` tale che `T1` è un sottotipo di `C` diverso da `T`; se `v` è vuoto deve quindi ritornare 0. Ad esempio, il seguente programma deve compilare e provocare le stampe indicate.

```cpp
int Fun(const vector<B*>& v){
        int cont = 0;
        if(v.isEmpty()) return 0;
        for(const vector::const_iterator cit = v.begin(); cit ≠ v.end(); ++cit){

        // for(auto it = v.begin(); it ≠ v.end(); ++it) - forma "normale"
            C* c = dynamic_cast<C*>(*it);
                if(c && typeid(*c) ≠ typeid(*v[0]){
                        cont++;
                }

        }
}
```

```cpp
#include<iostream>
#include<typeinfo>
#include<vector>
using namespace std;

class B {public: virtual ~B()
class C: public B {};
class D: public B {};
class E: public C {};

int Fun(vector<B*> &v){...}

main() {
  vector<B*> u, v, w;
  cout << Fun(u); // stampa 0
  B b; C c; D d; E e; B *p = &e, *q = &c;
  v.push_back(&c); v.push_back(&b); v.push_back(&d); v.push_back(&c);
  v.push_back(&e); v.push_back(p);
  cout << Fun(v); // stampa 2
  w.push_back(p); w.push_back(&d); w.push_back(q); w.push_back(&e);
  cout << Fun(w); // stampa 1
}
```

```cpp
int Fun(const vector<B*>& v) {
  int tot = 0;
  for(auto it = v.begin(); it!= v.end();
      ++it)
    if(typeid(*v[0])!= typeid(*(*it)) &&
       dynamic_cast<C*>(*it)) ++tot;
  return tot;
}
```

Esercizio Definizioni

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

class B {
private:
  Z bz;
};

class C: virtual public B {
private:
  Z cz;
};

class D: public C {
};

class E: virtual public B {
public:
  Z ez;
  // ridefinizione assegnazione
  // standard di E
};

class F: public D, public E {
private:
  Z* fz;
public:
  // ridefinizione del costruttore di copia profonda di F
  // ridefinizione del distruttore profondo di F
  // definizione del metodo di clonazione di F
};
```

Si considerino le definizioni sopra.
(1) Ridefinire l'assegnazione della classe `E` in modo tale che il suo comportamento coincida con quello dell'assegnazione standard di `E`. Naturalmente non è permesso l'uso della keyword `default`.
(2) Ridefinire il costruttore di copia profonda della classe `F`.
(3) Ridefinire il distruttore profondo della classe `F`.
(4) Definire il metodo di clonazione della classe `F`.

```
E& operator=(const E& e) {
  // ASSEGNAZIONE VARIABILI
  // RETURN *THIS
}
```

```
class E: virtual public B {
public:
  Z ez;
  // ridefinizione assegnazione
  // standard di E
};
```

```
E& operator=(const E& e){
  ez = e.ez;
  RETURN *THIS
}
```

ASSEGNAZIONE (PROFONDO) COPIA

```
E& operator=(const E& e);  <----------->  E(const E& e);
```

WHY?

PROFONDO (DEEP) → COPIA / ASSEGNA ANCHE TUTTO L'OGGETTO

(VECTOR<INT*>) :: ITERATOR

↑ STD ↑ TEMPLATE

```
E(const E& e){
}
```

→ RIFERIMENTO COSTANTE = NO SPRECO MEMORIA!

COSTRUTTORE <-> CLASSE (RIF) : (LISTA DI INIT. ...)

```
// parametri della classe: Z (int) ez;
E(const E& e): ez(e.ez) {};   -> COMPATTA

// oppure (scrivi la prima!)

E(const E& e) {
        ez = z.ez;   -> UGUALE, MA NON GRATIS
}

E& operator=(const E& e){
        // if (this != e) // NO per ereditarietà (SI in tutti gli altri casi)
        ez = z.ez;
        return *this;
}
```

E / ERR

* THIS

UGUALE MA DIVERSO RISPETTO ALLA COPIA!

```
// SOLUZIONE
class E: virtual public B {
public:
  Z ez;
  E& operator(const E& e) {
    B::operator=(e);
    ez=e.ez;
    return *this;
  }
};
```

PRIMO PUNTO! :)

```
class F: public D, public E {
private:
  Z* fz;
public:
  // ridefinizione del costruttore di copia profonda di F
  // ridefinizione del distruttore profondo di F
  // definizione del metodo di clonazione di F
};
```

F(const F& f): D(f), E(f), fz(f.fz);

F (D, E)

Si considerino le definizioni sopra.
(1) Ridefinire l'assegnazione della classe `E` in modo tale che il suo comportamento coincida con quello dell'assegnazione standard di `E`. Naturalmente non è permesso l'uso della keyword `default`.
(2) Ridefinire il costruttore di copia profonda della classe `F`.
(3) Ridefinire il distruttore profondo della classe `F`.
(4) Definire il metodo di clonazione della classe `F`.

→ F (CONST F& F):
FZ(F.FZ)

SEQUENZIONE

fz(f.fz≠nullptr ? new Z(*f.fz) : nullptr)
V F

SE ESISTE O NON

IF (F.FZ ≠ NULLPTR)
FZ = NEW Z (* F.FZ);
ELSE
FZ = NULLPTR

B, A
C
→ STANDARD

SOTTINTESI...

C C1 // A() B() C()

```
class F: public D, public E {
private:
  Z* fz;
public:
  // ridefinizione del costruttore di copia profonda di F
  // ridefinizione del distruttore profondo di F
  // definizione del metodo di clonazione di F
};
```

Si considerino le definizioni sopra.
(1) Ridefinire l'assegnazione della classe `E` in modo tale che il suo comportamento coincida con quello dell'assegnazione standard di `E`. Naturalmente non è permesso l'uso della keyword `default`.
(2) Ridefinire il costruttore di copia profonda della classe `F`.
(3) Ridefinire il distruttore profondo della classe `F`.
(4) Definire il metodo di clonazione della classe `F`.

~F() {
IF (FZ) DELETE FZ;
}

(3) `~F() {delete fz;}`

(4) CLONAZIONE

```
    return *this;
  }
};

class F: public D, public E {
private:
  Z* fz;
public:
  F(const F& f): B(f), D(f), E(f), fz(f.fz!=nullptr ? new Z(*f.fz) : nullptr) {}
  ~F() {delete fz;}
  virtual F* clone() const {return new F(*this);}
};
```

VIRTUAL TIPO * CLONE () CONST

RETURN NEW TIPO (* THIS);

← CLONA POLIMORFA

- IN SUMMARY...

```
CLASS B : PUBLIC A
    INT X

    // BOOL OPERATOR = (CONST BOOL B)
    A::OPERATOR = (B)
        X = B.X
    // B(CONST BOOL B) : A(B), X(B.X)
```