Definire un template di funzione `Fun(T1*, T2&)` che ritorna un booleano con il seguente comportamento. Consideriamo una istanziazione implicita `Fun(p,r)` dove supponiamo che i parametri di tipo `T1` e `T2` siano istanziati a tipi polimorfi (cioè che contengono almeno un metodo virtuale). Allora `Fun(p,r)` ritorna `true` se e soltanto se valgono le seguenti condizioni:

1. i parametri di tipo `T1` e `T2` sono istanziati allo stesso tipo;
2. siano `D1*` il tipo dinamico di `p` e `D2&` il tipo dinamico di `r`. Allora (i) `D1` e `D2` sono lo stesso tipo e (ii) questo tipo è un sottotipo proprio della classe `ios` della gerarchia di classi di I/O (si ricordi che `ios` è la classe base astratta della gerarchia).

```
template <class T1, class T2>
bool Fun(T1* p, T2& r) {
  return
    typeid(T1)==typeid(T2) &&
    typeid(*p)==typeid(r) &&
    dynamic_cast<ios*>(p);
}
```

Si consideri la gerarchia di classi per l'I/O. La classe base `ios` ha il distruttore virtuale, il costruttore di copia privato ed un unico costruttore (a 2 parametri con valori di default) protetto. Diciamo che le classi derivate da `istream` ma non da `ostream` (ad esempio `ifstream`), e `istream` stessa, sono *classi di input*, le classi derivate da `ostream` ma non da `istream` (ad esempio `ofstream`), ed `ostream` stessa, sono *classi di output*, mentre le classi derivate sia da `istream` che da `ostream` sono *classi di I/O* (esempi: `iostream` e `fstream`). Quindi ogni classe di input, output o I/O è una sottoclasse di `ios`. Definire una funzione `int F(ios& ref)` che restituisce -1 se il tipo dinamico di `ref` è un riferimento ad una classe di input, 1 se il tipo dinamico di `ref` è un riferimento ad una classe di output, 0 se il tipo dinamico di `ref` è un riferimento ad una classe di I/O, mentre in tutti gli altri casi ritorna 9.

```
int F(ios& ref) {
  if(dynamic_cast<istream*>(&ref) &&
     !dynamic_cast<ostream*>(&ref)) return -1;
  if(dynamic_cast<ostream*>(&ref) &&
     !dynamic_cast<istream*>(&ref)) return 1;
  if(dynamic_cast<istream*>(&ref) &&
     dynamic_cast<ostream*>(&ref)) return 0;
  return 9;
}
```

Esercizio

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.

```
int Fun(const std::vector<B*>& v){
        int tot = 0;
        for(auto it = v.begin(); it != v.end(); ++it){
                if(typeid(**v) != typeid(*(*it))
                && dynamic_cast<C*>(*it){
                        tot++;
                }
        }
        tot != 0 ? return tot : return 0;
        // ternary: (cond) ? true : false
}
```

Esercizio Costruttore

```
class A {
private:
  virtual void f() const =0;
  vector<int*>* ptr;
};
```

```
class D: virtual public A {
private:
  int z;
  double w;
};
```

```
class E: public D {
private:
  vector<double*> v;
  int* p;
  int& ref;
public:
   void f() const {}
   E(): p(new int(0)), ref(*p) {}
   // ridefinizione del costruttore di copia di E
};
```

Si considerino le precedenti definizioni. Ridefinire (senza usare la keyword `default`) nello spazio sottostante il costruttore di copia della classe `E` in modo tale che il suo comportamento coincida con quello del costruttore di copia standard di `E`.

ridefinizione del costruttore di copia di E

```
  // Copia (profonda)

  E(const E& e): D(e), v(e.v), (p == nullptr) ? p = e.p = nullptr : p = e.p, ref(e.ref) {}

  // Assegnazione (profonda)

  E& operator=(const E& e){
          D::operator=(e);
          // se no ereditarietà → if (this ≠ e) → Non siamo nela stessa zona memoria

          v = e.v; // analogo a v(e.v);
          p = e.p;
          ref = e.ref;

          return *this;
  }

  // Distruzione (profonda)

  ~E() {
          // p
          if(p) delete p;
          // vector
          for(auto it: v){
                  v.erase(it);
          }
  }
```