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).

Ad esempio, il seguente `main()` deve compilare e provocare le stampe indicate:

```
#include<iostream>
#include<fstream>
#include<typeinfo>
using namespace std;

class C { public: virtual ~C() {} };

main() {
  ifstream f("pippo"); fstream g("pluto"), h("zagor"); iostream* p = &h;
  C c1,c2;
  cout << Fun(&cout,cin) << endl; // stampa: 0
  cout << Fun(&cout,cerr) << endl; // stampa: 1
  cout << Fun(p,h) << endl; // stampa: 0
  cout << Fun(&f,*p) << endl; // stampa: 0
  cout << Fun(&g,h) << endl; // stampa: 1
  cout << Fun(&c1,c2) << endl; // stampa: 0
}
```

Si considerino i seguenti fatti concernenti la libreria di I/O standard.

- Si ricorda che `ios` è la classe base di tutta la gerarchia di classi della libreria di I/O, che la classe `istream` è derivata direttamente e virtualmente da `ios` e che la classe `ifstream` è derivata direttamente da `istream`.
- La classe base `ios` ha il distruttore virtuale. La classe `ios` rende disponibile un metodo costante e non virtuale `bool fail()` con il seguente comportamento: una invocazione `s.fail()` ritorna `true` se e solo se lo stream `s` è in uno stato di fallimento (cioè, il failbit di `s` vale 1).
- La classe `istream` rende disponibile un metodo non costante e non virtuale `long tellg()` con il seguente comportamento: una invocazione `s.tellg()`:
  1. se `s` è in uno stato di fallimento allora ritorna -1;
  2. altrimenti, cioè se `s` non è in uno stato di fallimento, ritorna la posizione della cella corrente di input di `s`.
- La classe `ifstream` rende disponibile un metodo non costante e non virtuale `bool is_open()` con il seguente comportamento: una invocazione `s.is_open()` ritorna `true` se e solo se il file associato allo stream `s` è aperto.

Definire una funzione `long Fun(const ios&)` con il seguente comportamento: una invocazione `Fun(s)`:

(1) se `s` è in uno stato di fallimento lancia una eccezione di tipo `Fallimento`; si chiede anche di definire tale classe `Fallimento`;

(2) se `s` non è in uno stato di fallimento allora:

   (a) se `s` non è un `ifstream` ritorna -2;

   (b) se `s` è un `ifstream` ed il file associato non è aperto ritorna -1;

   (c) se `s` è un `ifstream` ed il file associato è aperto ritorna la posizione della cella corrente di input di `s`.

Si consideri la seguente realtà concernente i biglietti del treno. Come ben noto, un biglietto per un viaggio in treno può essere di prima o seconda classe.

1. Definire una classe `Biglietto` i cui oggetti rappresentano un biglietto per un viaggio in treno. Ogni `Biglietto` è caratterizzato dalla distanza chilometrica del viaggio. La classe `Biglietto` dichiara un metodo virtuale puro `double prezzo()` che prevede il seguente contratto: una invocazione `b.prezzo()` ritorna il prezzo del biglietto `b`. Per tutti i biglietti, il prezzo base al km è fissato in 0.1 €.

2. Definire una classe `BigliettoPrimaClasse` derivata da `Biglietto` i cui oggetti rappresentano un biglietto per un viaggio di prima classe. Il prezzo di un biglietto di prima classe con distanza inferiore a 100 km è dato dal prezzo base (prezzo base al km moltiplicato per la distanza chilometrica) aumentato del 30%, altrimenti l'aumento del prezzo base è del 20%. `BigliettoPrimaClasse` implementa quindi `prezzo()` ritornando il prezzo di un dato biglietto di prima classe.

3. Definire una classe `BigliettoSecondaClasse` derivata da `Biglietto` i cui oggetti rappresentano un biglietto per un viaggio di seconda classe. Un biglietto di seconda classe può essere con prenotazione oppure senza (la prenotazione garantisce il posto a sedere). Per tutti i biglietti di seconda classe, il costo della prenotazione è fissato in 5 €. Il prezzo di un biglietto di seconda classe è dato dal prezzo base (prezzo base al km moltiplicato per la distanza chilometrica) più l'eventuale costo della prenotazione.

4. Definire una classe `BigliettoSmart` i cui oggetti rappresentano dei puntatori smart a `Biglietto`. La classe `BigliettoSmart` dovrà essere dotata dell'interfaccia pubblica necessaria per lo sviluppo della successiva classe `Treno`.

5. Definire una classe `TrenoPieno` i cui oggetti rappresentano delle eccezioni che segnalano che non vi sono posti disponibili in un dato treno. Una eccezione di `TrenoPieno` è caratterizzata dalla classe (1^ o 2^) in cui non vi sono più posti disponibili.

6. Definire una classe `Treno` i cui oggetti rappresentano un certo viaggio in treno (la semplificazione prevede che non vi siano fermate intermedie). Ogni oggetto `Treno` è quindi caratterizzato dall'insieme dei biglietti venduti per quel viaggio in treno, e tale insieme deve essere rappresentato mediante un vector `venduti` di puntatori smart `BigliettoSmart`. Un oggetto `Treno` è inoltre caratterizzato dal numero massimo di posti disponibili per biglietti di prima classe e dal numero massimo di posti disponibili per biglietti di seconda classe con prenotazione.

   Devono essere disponibili nella classe `Treno` le seguenti funzionalità:

   - Un metodo `int* bigliettiVenduti()` con il seguente comportamento: una invocazione `t.bigliettiVenduti()` ritorna un array `ar` di 3 interi tale che:
     - `ar[0]` memorizza il numero di biglietti venduti di prima classe per il treno `t`;
     - `ar[1]` memorizza il numero di biglietti venduti di seconda classe con prenotazione per il treno `t`;
     - `ar[2]` memorizza il numero di biglietti venduti di seconda classe senza prenotazione per il treno `t`.
   - Un metodo `void vendiBiglietto(const Biglietto&)` con il seguente comportamento: una chiamata `t.vendiBiglietto(b)` aggiunge `b` tra i biglietti venduti per il treno `t` quando possibile, altrimenti solleva una opportuna eccezione di `TrenoPieno`. Più dettagliatamente:
     - Se `b` è un biglietto di prima classe e vi sono ancora posti di prima classe disponibili in `t` allora viene aggiunto al vector `venduti` un puntatore smart a `b`; se invece non vi sono posti di prima classe disponibili viene sollevata una eccezione `TrenoPieno` in prima classe.
     - Se `b` è un biglietto di seconda classe con prenotazione e vi sono ancora posti di seconda classe con prenotazione disponibili in `t` allora viene aggiunto al vector `venduti` un puntatore smart a `b`; se invece non vi sono posti di seconda classe con prenotazione disponibili viene sollevata una eccezione `TrenoPieno` in seconda classe.
     - Se `b` è un biglietto di seconda classe senza prenotazione allora viene sempre aggiunto al vector `venduti` un puntatore smart a `b`.
   - Un metodo `double incasso()` con il seguente comportamento: una chiamata `t.incasso()` ritorna l'incasso totale per tutti i biglietti sinora venduti per il treno `t`.