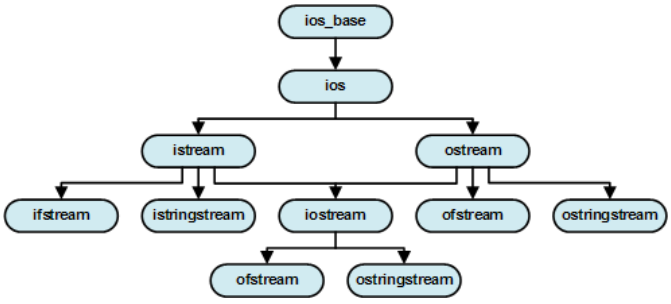
Definire un template di funzione
`template <class T> list<const iostream*> compare(vector<ostream*>&, vector<const T*>&)`
con il seguente comportamento: in ogni invocazione `compare(v,w)`,

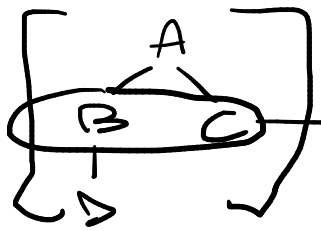
1. se `v` e `w` non contengono lo stesso numero di elementi allora viene sollevata una eccezione di tipo `string` che rappresenta la stringa vuota;
2. se `v` e `w` contengono lo stesso numero di elementi allora per ogni posizione `i` dentro i bounds dei due vettori `v` e `w`:
   (a) se `*v[i]` è un `fstream` ed è dello stesso tipo di `*w[i]` allora: (i) il puntatore `v[i]` viene inserito nella lista che la funzione deve ritornare; (ii) i puntatori `v[i]` e `w[i]` vengono rimossi dai vettori che li contengono;
   (b) se `*w[i]` è uno `stringstream` in stato good e `*v[i]` e `*w[i]` sono di tipo diverso allora il puntatore `w[i]` viene inserito nella lista che la funzione deve ritornare.

```
template<class T>
list<const iostream*> compare(vector<ostream*> &v, vector<const T*> &w){

        // size = dimensione - capacity = elementi attualmente allocati nel vettore
        if(v.size() != w.size()){
                throw std::string("");
        }

        list<const iostream*> list;

        // for(vector<ostream*>::iterator it  ... - v/w con stessa size, prendiamo v!

        // nota: "for" con più contatori, si scrive così ...
        // nota2: incremento prefisso leggermente più efficiente
        for(auto it = v.begin(), it2 = w.begin(); it != v.end(); ++it, ++it2){
                fstream *f = dynamic_cast<fstream*>(*it); // ostream

                // fstream* f = dynamic_cast<fstream*>(const_cast<ios*>)(*it)); const ostream

                if(f && typeid(*f) == typeid(T)){

                        // push_back = inserimento casuale (efficiente)
                        // mettere i valori nel vector
                        // push_front (equivalente)

                        list.push_back(f);

                        // it = v.erase(it); <----> v.erase(it) - equivalenza
                        // N.B. di solito quando non const! (const-correctness)

                        v.erase(f); // rimuovi da v (come visto)

                        T *t = const_cast<T*>(*it2);    // togli const da w[i];
                        w.erase(t);
                }

                stringstream* s = dynamic_cast<stringstream*>(*it2);
                if(s && typeid(*s) != typeid(*f) && s->good()){
                        list.push_back(s);
                }
        }

        return list;
}
```



Siano A, B, C e D quattro diverse classi polimorfe. Si considerino le seguenti definizioni:

```
template <class X, class Y>
X* fun(X* p) { return dynamic_cast<Y*>(p); }

main(){
     C c;
     fun <A,B> ( &c );
     if  ( fun <A,B> (new C( ) ) = = 0 )
          cout<<"Programazione";
     if  ( dynamic_cast <C*> ( new B( ) ) = = 0 )
          cout<<"ad oggetti";
     A* p = fun <D,B> (new D( ) );
}
```



Si supponga che:

1. il *main( )* compili correttamente ed esegua senza provocare errori a run time;
2. l'esecuzione del *main( )* provochi in output su *cout* la stampa "Programmazione ad oggetti".

In tali ipotesi, per ognuna delle relazioni di sottotipo *T1<=T2* nelle seguenti tabelle segnare:

1. VERO per indicare che T1 **sicuramente** è sottotipo di T2;
2. FALSO per indicare che T1 **sicuramente non** è sottotipo di T2;
3. POSSIBILE altrimenti.

A<=B	.....
A<=C	.....
A<=D	.....
B<=A	.....
B<=C	.....
B<=D	.....
C<=A	.....
C<=B	.....
C<=D	.....
D<=A	.....
D<=B	.....
D<=C	.....

```
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 `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