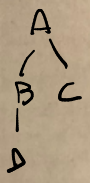
Esercizio 4

```
class Z {
public:
  Z(int x=0) {}
};

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;
};

class F: public D, public E {
private:
  Z* pz;
public:
  // ridefinizione del costruttore di copia di F
};
```

```
// 1. Copia profonda

F(const F& f): D(f), E(f), pz(f.pz) {}
// (pz == nullptr) ? pz = nullptr : pz = new F(*f.pz);
// OCCHIO PER RANZY: se il campo è un puntatore lo inizializzi e consideri il caso vuoto

// 2. Assegnazione profonda

F& operator=(const F& f){
        // if(this ≠ &f) → Controllo SOLO nelle modellazioni → garantito per ereditarietà

        D::operator(f);
        E::operator(f);
        pz = f.pz;
        // Masochismo: (pz == nullptr) ? pz = nullptr : pz = new F(*f.pz);

        return *this;
}

// 3. Clonazione (polimorfa)
virtual F* clone() const { return new F(*this); }
```

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

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

main() {
  C c; fun<A,B>(&c);
  if( fun<A,B>(new C()) == 0 ) cout << "Bjarne ";
  if( dynamic_cast<C*>(new B()) == 0 ) cout << "Stroustrup";
  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 `Bjarne Stroustrup`.

In tali ipotesi, per ognuna delle relazioni di sottotipo `T1`≤`T2` nelle seguenti tabelle segnare con una croce l'entrata

(a) "Vero" per indicare che `T1` **sicuramente è** sottotipo di `T2`;
(b) "Falso" per indicare che `T1` **sicuramente non è** sottotipo di `T2`;
(c) "Possibile" **altrimenti**, ovvero se non valgono nè (a) nè (b).

	Vero	Falso	Possibile
A≤B		X	
A≤C		X	
A≤D		X	
B≤A	X		
B≤C			X
B≤D		X	

	Vero	Falso	Possibile
C≤A	X		
C≤B			X
C≤D		X	
D≤A	X		
D≤B	X		
D≤C		X	

```
#include<iostream>
#include<string>
using namespace std;

class Z {
public:
  operator int() const {return 0;}
};

template<class T> class D; // dichiarazione incompleta

template<class T1, class T2 = Z, int k = 1>
class C {
  friend class D<T1>;
private:
  T1 t1;
  T2 t2;
  int a;
  C(int x =k): a(x) {}
};

template<class T>
class D {
 public:
  void f() const {C<T,T> c(1); cout << c.t1 << c.t2 << c.a;}
  void g() const {C<int> c;}
  void h() const {C<T,int> c(3); cout << c.t2 << c.a;}
  void m() const {C<int,T,3> c; cout << c.t1;}
  void n() const {C<int,double> c; cout << c.t1 << c.t2 << c.a;}
  void o() const {C<char,double> c(6); cout << c.a;}
  void p() const {C<Z,T,7> c(7); cout << c.t2 << c.a;}
};
```

Determinare se i seguenti `main()` compilano correttamente o meno barrando la corrispondente scritta.

```
int main() { D<char> d1; d1.f(); }
int main() { D<std::string> d2; d2.f(); }
int main() { D<char> d3; d3.g(); }
int main() { D<int> d4; d4.g(); }
int main() { D<char> d5; d5.h(); }
int main() { D<int> d6; d6.h(); }
int main() { D<char> d7; d7.m(); }
int main() { D<int> d8; d8.m(); }
int main() { D<char> d9; d9.n(); }
int main() { D<Z> d10; d10.n(); }
int main() { D<char> d11; d11.o(); }
int main() { D<Z> d12; d12.o(); }
int main() { D<char> d13; d13.p(); }
int main() { D<Z> d14; d14.p(); }
```

TEMPLATES → FUN(FUN<A>(PAT))

TD

TS → TIPO STATICO

TIPO DINAMICO

Esercizio Gerarchia [NB: scrivere la soluzione chiaramente nello spazio apposito e definire tutti i metodi inline]

Si consideri il seguente modello di realtà concernente il cloud software PaODrive® che fornisce un servizio di file hosting.

1. Definire la seguente gerarchia di classi.
   (a) Definire una classe `File` i cui oggetti rappresentano un file memorizzabile in un account PaODrive®. Ogni `File` è caratterizzato dalla sua dimensione (in MB) e dall'essere pubblicamente accessibile o meno. Dotare la classe `File` di opportuno/i costruttore/i.
   (b) Definire una classe `Photo` derivata da `File` i cui oggetti rappresentano un file che memorizza una immagine. Ogni `Photo` è caratterizzata dall'essere memorizzata in un formato *lossy* (compressione con perdita di informazione) oppure no. Dotare la classe `Photo` di opportuno/i costruttore/i.
   (c) Definire una classe `Video` derivata da `File` i cui oggetti rappresentano un file che memorizza un video. Ogni `Video` è caratterizzato dal suo *framerate* (un numero possibilmente decimale che rappresenta la frequenza dei fotogrammi). Dotare la classe `Video` di un costruttore che includa un parametro formale `f` di tipo `double` per costruire un `Video` con framerate `f` (se `f` < 0 allora il framerate è impostato a zero).
2. Definire una classe `PaODrive` i cui oggetti rappresentano un account PaODrive®. Più precisamente, un oggetto `PaODrive` è caratterizzato dai file memorizzati dall'account PaODrive®, che sono rappresentati mediante un contenitore di puntatori al tipo `File`, e dalla capacità massima di memorizzazione (in MB) dell'account PaODrive®. Devono essere disponibili le seguenti funzionalità:
   (a) un metodo `double uploadFile(const File&)` con il seguente comportamento: una invocazione `pao.uploadFile(f)` provoca il caricamento (upload) del file `f` qualora la memoria residua dell'account PaODrive® `pao` lo consenta e quindi ritorna lo spazio di memorizzazione rimasto disponibile per l'account `pao` dopo tale upload; se la memoria residua dell'account PaODrive® `pao` non consenta l'upload allora il file `f` non viene caricato in `pao` e viene ritornato il valore -1.0.
   (b) un metodo `vector<const File*> copy(double)` con il seguente comportamento: una invocazione `pao.copy(x)` ritorna un vector (eventualmente vuoto) contenente i puntatori a:
      (i) tutti i video memorizzati nell'account `pao` che: (1) sono pubblicamente accessibili e (2) hanno un framerate $\geq$ `x`.
      (ii) tutte le immagini memorizzati nell'account `pao` che: (1) non sono in un formato lossy e (2) hanno dimensione minore a 2 MB.

   Nella definizione di tale metodo **non è possibile** usare l'operatore di indicizzazione su qualsiasi contenitore.

```
class File{
        private:
                unsigned int size;
                bool isAccessible;
        public:
                File(unsigned int s, bool isA): size(s), isAccessible(isA) {};
                virtual ~File() {};
};

// lossy o meno
class Photo: public File{
        private:
                bool isLossy;
        public:
                Photo(unsigned int s, bool isA, bool isL): File(s, isA), isLossy(isL) {};
};

// framerate: se < 0, framerate a 0, oppure framerate = f
// parametro formale ⟶ quello della firma della funzione (prototipo) ...

class Video: public File{
        private:
                double framerate;
        public:
                Video(unsigned int s, bool isA, double f): File(s, isA),
                        (f < 0) ? framerate = 0 : framerate = f {
                        // alternativa safe ad usare operatore ternario ⟶ (condizione) ? vero : falso
                        // if(f < 0) framerate = 0;
                        // else framerate = f;
                };
};
```

2. Definire una classe `PaODrive` i cui oggetti rappresentano un account PaODrive®. Più precisamente, un oggetto `PaODrive` è caratterizzato dai file memorizzati dall'account PaODrive®, che sono rappresentati mediante un contenitore di puntatori al tipo `File`, e dalla capacità massima di memorizzazione (in MB) dell'account PaODrive®. Devono essere disponibili le seguenti funzionalità:

   (a) un metodo `double uploadFile(const File&)` con il seguente comportamento: una invocazione `pao.uploadFile(f)` provoca il caricamento (upload) del file `f` qualora la memoria residua dell'account PaODrive®`pao` lo consenta e quindi ritorna lo spazio di memorizzazione rimasto disponibile per l'account `pao` dopo tale upload; se la memoria residua dell'account PaODrive®`pao` non consenta l'upload allora il file `f` non viene caricato in `pao` e viene ritornato il valore -1.0.

   (b) un metodo `vector<const File*> copy(double)` con il seguente comportamento: una invocazione `pao.copy(x)` ritorna un vector (eventualmente vuoto) contenente i puntatori a:

      (i) tutti i video memorizzati nell'account `pao` che: (1) sono pubblicamente accessibili e (2) hanno un framerate $\geq$ `x`.

      (ii) tutte le immagini memorizzati nell'account `pao` che: (1) non sono in un formato lossy e (2) hanno dimensione minore a 2 MB.

   Nella definizione di tale metodo **non è possibile** usare l'operatore di indicizzazione su qualsiasi contenitore.

```
// list vs vector ⟶ vector è più efficiente (usa quello, ma non cambia nulla..)
class PaoDrive{
        private:
                vector<File*> pao; ⟶ quello che capisco dal testo come tipo..

                // vector<const File*> pao;  ⟶ consiglio perché i punti dopo te lo dicono ...

                unsigned int capacity;
        public:
                double uploadFile(const File& f){
                        if(capacity - f.size() > 0){
                                pao.push_back(f); // ancora spazio ...
                                return capacity - f.size();
                        }
                        else return -1.0;
                }

                vector<const File*> copy(double x){
                        vector<const File*> ret;

                        for(auto it = pao.begin(); it ≠ v.end; ++it){
                                // video pubblicamente accessibili e con framerate ≥ x;

                                Video* v = dynamic_cast<Video*>(const_cast<File*>(*it));
                                if(v && v→isAccessible() && v→getFramerate() ≥ x){
                                        pao.push_back(dynamic_cast<const Video*>(*v));
                                        // il vector di ritorno è const, qua BISOGNA convertire
                                        // al tipo const

                                        // OCCHIO
                                        // vector contenente le "copie" dei puntatori a video ...
                                        // pao.push_back(v→clone());
                                }

                                // immagini non lossy e size ≤ 2 MB
                                Photo* p = dynamic_cast<Photo*>(const_cast<File*>(*it));
                                if(p && !p→isLossy() && p→size() ≤ 2){
                                        pao.push_back(dynamic_cast<const Photo*>(*p));

                                        // il vector di ritorno è const, qua BISOGNA convertire
                                        // al tipo const
                                }
                        }

                        return ret;
                }
};
```