Quesito 1:

quesito1.cpp

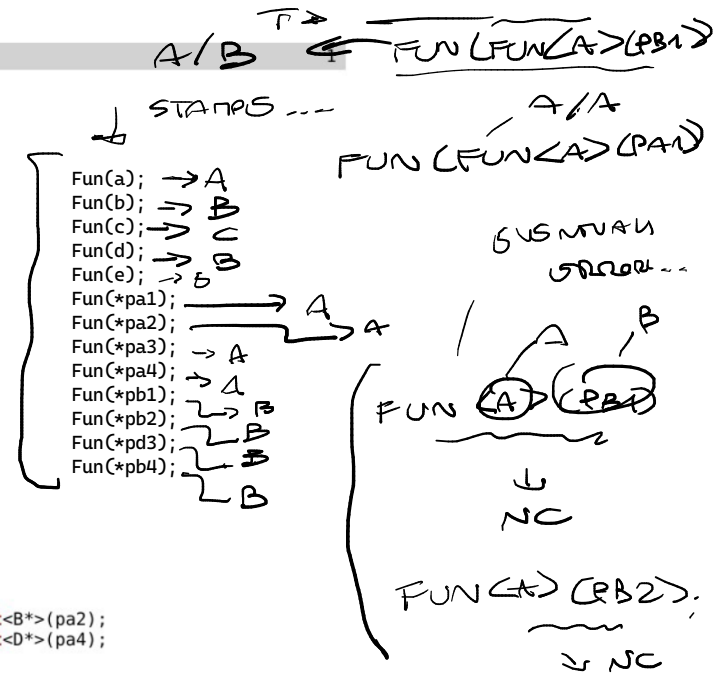
```cpp
#include <iostream>

using namespace std;

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

template <class T>
void Fun(T* pt){
    bool b=0;
    try{ throw T(*pt); }
    catch(E){cout<<"E";b=1;}
    catch(B){cout<<"B";b=1;}
    catch(D){cout<<"D";b=1;}
    catch(C){cout<<"C";b=1;}
    catch(A){cout<<"A";b=1;}
    if(!b) cout<<"NO";
}


main(){
A a;B b;C c;D d;E e;
A* pa1=&b, *pa2=&c,*pa3=&d,*pa4=&e;
B* pb1=dynamic_cast<B*>(pa1); B* pb2=dynamic_cast<B*>(pa2);
B* pd3=dynamic_cast<D*>(pa3); B* pb4=dynamic_cast<D*>(pa4);
}
```



Esercizio 2

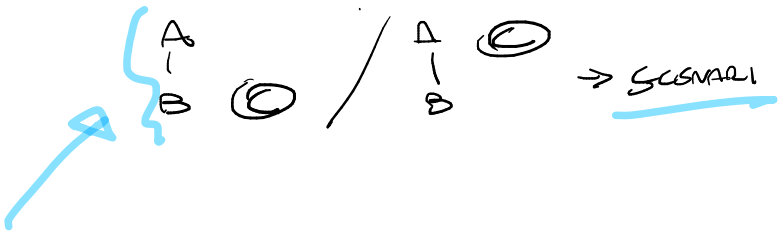
Scrivere un programma che consista esattamente di tre classi `A`, `B` e `C`, dove `B` è un sottotipo di `A`, mentre `C` non è in relazione di subtyping nè con `A` nè con `B`, che dimostri in un metodo di `C` un tipico esempio di un **uso giustificato e necessario** della conversione di tipo `dynamic_cast` per effettuare type downcasting. A questo fine, si usino il **minor numero possibile di metodi**.

```cpp
class A{};

class B: public A{};

class C{};

int main{
        // type downcasting
        A a;
        B* b = dynamic_cast<B*>(a);
        C* c = dynamic_cast<C*>(b);
}
```



Esercizio 3 C++

Definire una unica gerarchia di classi che includa:

(1) una classe base polimorfa `A` alla radice della gerarchia;

(2) una classe derivata astratta `B`;

(3) una sottoclasse `C` di `B` che sia concreta;

(4) una classe `D` che non permetta la costruzione pubblica dei suoi oggetti, ma solamente la costruzione di oggetti di `D` che siano sottooggetti;

(5) una classe `E` derivata direttamente da `D` e con l'assegnazione ridefinita pubblicamente con comportamento identico a quello dell'assegnazione standard di `E`.

NB: Scrivere la soluzione **chiaramente** nel foglio a quadretti.

STANDARDS...

```cpp
class A{
        virtual ~A();
}

class B: public A{
        public:
                virtual int method() = 0;
};
```

```cpp
class C: public B{
        public:
                virtual int method() {
                        return 42;
                }
}

class D{
        private: int x;
        protected:
                D(int x1): x(x1) {}
};
```

```cpp
class E: public D{
        E& operator=(const E& e){
                D::operator=(e);
                x = e.x;
                return *this;
        }
};
```

Si considerino le seguenti definizioni di classe e funzione:

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

char F(A* p, C& r) {
  B* punt1 = dynamic_cast<B*> (p);
  try{
    E& s = dynamic_cast<E&> (r);
  }
  catch(bad_cast) {
    if(punt1) return 'O';
    else return 'M';
  }
  if(punt1) return 'R';
  return 'A';
}
```

R ≡ F(B, B)
O ≡ F(B, C)
M ≡ F(A, C)
A ≡ F(A, B)

Si consideri inoltre il seguente `main()` incompleto, dove `?` è semplicemente un simbolo per una incognita:

```
main(){
  A a; B b; C c; D d; E e;
  cout << F(?,?) << F(?,?) << F(?,?) << F(?,?);
}
```

Definire opportunamente le chiamate in tale `main()` usando gli oggetti `a`, `b`, `c`, `d`, `e` locali al `main()` in modo tale che la sua esecuzione provochi la stampa `ROMA`.