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
};
```

(1) (2)

① QUALI CLASSI DERIVO

② QUANTI CAMPI HO

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(e.p), ref(e.ref) {}
```

ASSEGNAZIONE PROFONDA →

```
E& operator=(const E& e){
  D::operator=(e);
  v = e.v;
  p = e.p;
  ref = e.ref;
  return *this;
}
```

ESEMPIO CALCO → 2° ESAME 2021 ...

Esercizio Definizioni

```
class Z {
private:
  int x;
};

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;
  // ridefinizione assegnazione
  // standard di E
};
    E& operator=(const E& e){
      B::operator=(e);
      ez = e.ez;
      return *this;
    }

class F: public D, public E {
private:
  Z* fz;
public:
  // ridefinizione del costruttore di copia profonda di F
  // ridefinizione del distruttore profondo di F
  // definizione del metodo di clonazione di F
};
```

Si considerino le definizioni sopra.
(1) Ridefinire l'assegnazione della classe `E` in modo tale che il suo comportamento coincida con quello dell'assegnazione standard di `E`. Naturalmente non è permesso l'uso della keyword `default`.
(2) Ridefinire il costruttore di copia profonda della classe `F`.
(3) Ridefinire il distruttore profondo della classe `F`.
(4) Definire il metodo di clonazione della classe `F`.

```
// SOLUZIONE
class E: virtual public B {
public:
  Z ez;
  E& operator(const E& e) {
    B::operator=(e);
    ez=e.ez;
    return *this;
  }
};
```

F(const F& f): D(f), E(f), fz(f.fz) {}

↑ INCOMPLETO: È UN PUNTATORE!

BASE VIRTUALE ...

```
F(const F& f): B(f), D(f), E(f), fz(f.fz!=nullptr ? new Z(*f.fz) : nullptr) {}
~F() {delete fz;}
virtual F* clone() const {return new F(*this);}
```

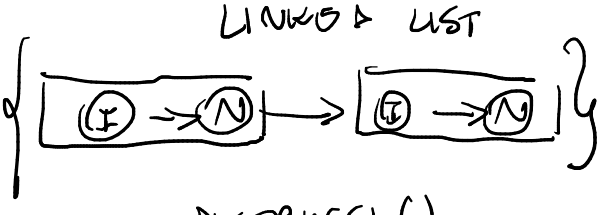
→ CLONAZIONE

DISTRUZIONE PROFONDA: IF (FZ) DELETE FZ ≡ DELETE FZ

↓

PERCHÉ DISTRUGGI ANCHE TUTTI I SOTTOCAMPI (LE CI SONO)

LINKED LIST



DISTRUGGI C.I. ....

CLASS C ——— NOME CLASSE CLONE()

Clonazione: `virtual C* clone() const { return new C*(this) };`

CLASS F : PUBLIC D, PUBLIC E, PUBLIC B

INT X;

Copia ⟶ `F(const F& f): D(f), E(f), B(f), x(f.x) {}`

Assegnazione ⟶
```
F& operator=(const F& f){
    D::operator=(f);
    E::operator=(f);
    B::operator=(f);

    x = f.x;

    return *this;
}
```
STANDARD...

Scrivere un template di classe `SmartP<T>` di **puntatori smart** a `T` che definisca assegnazione profonda, costruzione di copia profonda e distruzione profonda di puntatori smart. Il template `SmartP<T>` dovrà essere dotato di una interfaccia pubblica che permetta di **compilare correttamente** il seguente codice, la cui esecuzione dovrà **provocare esattamente** le stampe riportate nei commenti.

```
class C {
public:
  int* p;
  C(): p(new int(5)) {}
};

int main() {
  const int a=1; const int* p=&a;
  SmartP<int> r; SmartP<int> s(&a); SmartP<int> t(s);
  cout << *s << " " << *t << " " << *p << endl; // 1 1 1
  *s=2; *t=3;
  cout << *s << " " << *t << " " << *p << endl; // 2 3 1
  r=t; *r=4;
  cout << *r << " " << *s << " " << *t << " " << *p << endl; // 4 2 3 1
  cout << (s==t) << " " << (s!=p) << endl; // 0 1
  C c; SmartP<C> x(&c);
  cout << *(c.p) << " " << *(x->p) << endl; // 5 5
  *(c.p)=6;
  cout << *(c.p) << " " << *(x->p) << endl; // 6 6
  SmartP<C>* q = new SmartP<C>(&c);
  delete q;
  cout << *(x->p) << endl; // 6
}
```

OPERATOR== , OPERATOR!=] BOOL

IF (C!=R.C) TRUE ELSE FALSE

li esercizi > Tutte le tipologie > Modellazioni > Smartp > Soluzione.cpp > SmartP<T>

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

template <class T>
class SmartP{
    private:
        T* p;
    public:
        //costruttore di default
        SmartP(): p(nullptr) {}

        //costruttore ad un parametro
        SmartP(const T* punt): p(new T(*const_cast<T*>
        (punt))){}

        //costruttore di copia profondo
        SmartP(const SmartP<T>& s):p(new T(*(s.p))){}

        operator const T* () const {return p;}

        //distruttore profondo
        ~SmartP(){
          if (p)  delete p;
        }
        //assegnazione profonda
        SmartP<T>& operator=(const SmartP<T>& s){
```

C: > Users > roves > OneDrive > Documenti > GitHub > Informatica-UniPD > Corsi > 2 Anno

```cpp
class SmartP{
        SmartP<T>& operator=(const SmartP<T>& s){
            if(this!=&s)
            return *this;
        }

        //operatore di chiamata a funzione
        SmartP<T> operator ()(T*& p){return SmartP(p);}

        //operatore di dereferenziazione
        T& operator*() const{
            return p;
        }
        //operatore di selezione di membro
        T* operator->() const{
            return &p;
        }
};

class C{
    public:
        int *p;
        C(): p(new int (5)){}
};

int main(){
```

RIFERIMENTI COSTANTI → NO ALIASING = CONDIZIONI DI MEMORIA

Esercizio Funzione

```
class A {
public:
  virtual A* f() const =0;
};

class B: public A {};

class C: public B {
public:
  B* f() const {return new C();}
};

class D: public B {};

class E: public B {
public:
  A* f() const {return new E();}
};

class F: public C, public D, public E {
public:
  D* f() const {return new F();}
};
```

Queste definizioni compilano correttamente. Definire una funzione

`list<const D *const> fun(const vector<const B*>&)`

con il seguente comportamento: in ogni invocazione `fun(v)`, per tutti i puntatori `q` contenuti nel vector `v`:

(A) se `q` non è nullo ed ha un tipo dinamico esattamente uguale a `C` allora `q` deve essere rimosso da `v`;

(A_1) se il numero N di puntatori rimossi dal vector `v` è maggiore di 2 allora viene sollevata una eccezione di tipo `C`.

(B) sul puntatore `q` non nullo deve essere invocata la funzione virtuale pura `A* A::f()` che ritorna un puntatore che indichiamo qui con `ptr`;

(B_1) se `ptr` è nullo allora viene sollevata una eccezione `std::string("nullptr");`

(B_2) `fun` ritorna la lista di tutti e soli questi puntatori `ptr` che: non sono nulli e hanno un tipo dinamico che è sottotipo di `D*` e non è un sottotipo di `E*`.

```
list<const D* const> fun(const vector<const B*>& v){
        list<const D* const> l;
        int cont = 0;

        for(auto q = v.begin(); q != v.end(); ++q){
                if(q && typeid(*q) == typeid(C)){

                //q -> const vector<const B*>::const_iterator (!)

                B* b = const_cast<B*>(*q); // togliamo il const
                v.erase(b); // oppure q = v.erase(q);
                cont ++;
                if(cont > 2) throw C();
                }
                if(q) {
                        A* ptr = q->f();
                        if(ptr == std::nullptr)
                                throw std::string("nullptr");
                        if(ptr && dynamic_cast<D*>(*ptr) &&
                        !dynamic_cast<E*>(*ptr))

                                l.push_back(ptr);
                }
        }

        return l;
}
```

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:

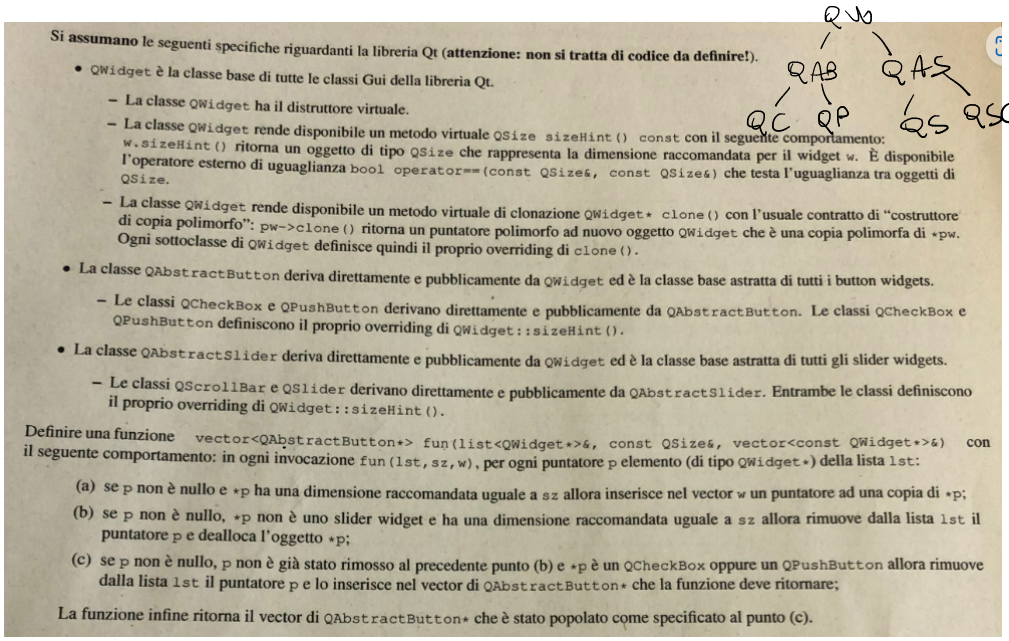
   (a) se `s` non è un `ifstream` ritorna -2;

   (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`.

```
long Fun(const ios& s){
        if(s.fail()) throw Fallimento("Fail");
        ifstream *i = dynamic_cast<ifstream*>(s);
        if(!i) return -2;
        if(i && !i->isOpen()) return -1;
        if(i && i->isOpen()) return i->tellg();
}

class Fallimento{
        private:
                std::string msg;
        public:
                Fallimento(std::string m):
                        msg(m) {};
};
```



Si assumano le seguenti specifiche riguardanti la libreria Qt (**attenzione: non si tratta di codice da definire!**).

- `QWidget` è la classe base di tutte le classi Gui della libreria Qt.
  - La classe `QWidget` ha il distruttore virtuale.
  - La classe `QWidget` rende disponibile un metodo virtuale `QSize sizeHint() const` con il seguente comportamento: `w.sizeHint()` ritorna un oggetto di tipo `QSize` che rappresenta la dimensione raccomandata per il widget `w`. È disponibile l'operatore esterno di uguaglianza `bool operator==(const QSize&, const QSize&)` che testa l'uguaglianza tra oggetti di `QSize`.
  - La classe `QWidget` rende disponibile un metodo virtuale di clonazione `QWidget* clone()` con l'usuale contratto di "costruttore di copia polimorfo": `pw->clone()` ritorna un puntatore polimorfo ad nuovo oggetto `QWidget` che è una copia polimorfa di `*pw`. Ogni sottoclasse di `QWidget` definisce quindi il proprio overriding di `clone()`.
- La classe `QAbstractButton` deriva direttamente e pubblicamente da `QWidget` ed è la classe base astratta di tutti i button widgets.
  - Le classi `QCheckBox` e `QPushButton` derivano direttamente e pubblicamente da `QAbstractButton`. Le classi `QCheckBox` e `QPushButton` definiscono il proprio overriding di `QWidget::sizeHint()`.
- La classe `QAbstractSlider` deriva direttamente e pubblicamente da `QWidget` ed è la classe base astratta di tutti gli slider widgets.
  - Le classi `QScrollBar` e `QSlider` derivano direttamente e pubblicamente da `QAbstractSlider`. Entrambe le classi definiscono il proprio overriding di `QWidget::sizeHint()`.

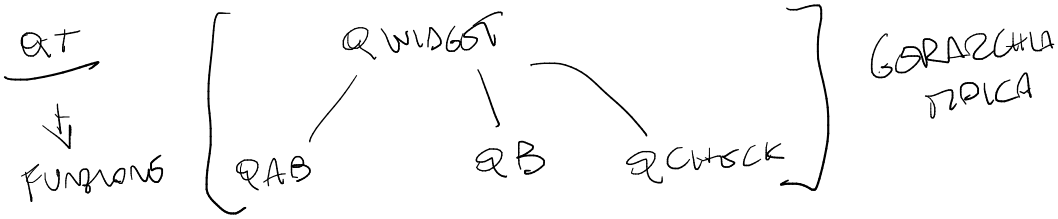
Definire una funzione `vector<QAbstractButton*> fun(list<QWidget*>&, const QSize&, vector<const QWidget*>&)` con il seguente comportamento: in ogni invocazione `fun(lst,sz,w)`, per ogni puntatore `p` elemento (di tipo `QWidget*`) della lista `lst`:

(a) se `p` non è nullo e `*p` ha una dimensione raccomandata uguale a `sz` allora inserisce nel vector `w` un puntatore ad una copia di `*p`;

(b) se `p` non è nullo, `*p` non è uno slider widget e ha una dimensione raccomandata uguale a `sz` allora rimuove dalla lista `lst` il puntatore `p` e dealloca l'oggetto `*p`;

(c) se `p` non è nullo, `p` non è già stato rimosso al precedente punto (b) e `*p` è un `QCheckBox` oppure un `QPushButton` allora rimuove dalla lista `lst` il puntatore `p` e lo inserisce nel vector di `QAbstractButton*` che la funzione deve ritornare;

La funzione infine ritorna il vector di `QAbstractButton*` che è stato popolato come specificato al punto (c).




```cpp
#include <iostream>
#include <vector>

using namespace std;
vector<QAbstractButton*> fun(list<QWidget*>&q, const QSize& sz, vector <const QWidget*>& v){
   vector<QAbstractButton*> v;
   list<QWidget*>::iterator it = q.begin();

   for(; it!=v.end(); ++it){
       if(*it && (*it)->sizeHint() == sz) q.push_front((*it)->clone());
       QAbstractSlider *s=dynamic_cast<QAbstractSlider*>(*it);
       if((*it) && !s && (*it)->sizeHint() == s){
           delete *it;
           it = lista.erase(it);
       }
       if((*it) && dynamic_cast<QCheckBox*>(*it) || dynamic_cast<QPushButton*>(*it)){
          v.push_back(static_cast<QAbstractButton*>(*it));
          it=lista.erase(it);
       }
   }
   return v;
}

```