Laurea in Informatica – Programmazione ad Oggetti – Appello d'Esame 28/01/2025

Nome..... Cognome..... Matricola.....

Esercizio Cosa Stampa

```
class A {
public:
  A() {cout<< " A() ";}
  virtual ~A() {cout<< " ~A() ";}
  virtual void f() {cout <<" A::f "; h(); g();}
  virtual void g() const {cout <<" A::g ";}
  virtual const A* h() {cout<<" A::h "; return this;}
  virtual void k() {cout <<" A::k "; m(); h(); }
  void m() {cout <<" A::m "; g(); h();}
  virtual A& n() {cout <<" A::n "; return *this;}
};

class C: virtual public A {
public:
  C() {cout<< " C() ";}
  ~C() {cout<< " ~C() ";}
  virtual void g() const override {cout <<" C::g ";}
  void k() override {cout <<" C::k "; A::n();}
  virtual void m() {cout <<" C::m "; h(); g();}
  A& n() override {cout <<" C::n "; return *this;}
};

class D: virtual public A {
public:
  D() {cout<< " D() ";}
  ~D() {cout<< " ~D() ";}
  virtual void g() {cout <<" D::g ";}
  const A* h() {cout <<" D::h "; return this;}
  void k() const {cout <<" D::k "; k();}
  void m() {cout <<" D::m "; g(); h();}
};

class E: public C, public D  {
public:
  E() {cout<< " E() ";}
  ~E() {cout<< " ~E() ";}
  virtual void g() const {cout <<" E::g ";}
  const E* h() {cout <<" E::h "; return this;}
  void m() {cout <<" E::m "; h(); g(); }
  D& n() final {cout <<" E::n "; return *this;}
};

class F: public E {
public:
  F() {cout<< " F() ";}
  ~F() {cout<< " ~F() ";}
  F(const F& x): A(x) {cout<< " Fc ";}
  void k() {cout <<" F::k "; h();}
  void m() {cout <<" F::m "; g();}
};

A* p1 = new E(); A* p2 = new C(); A* p3 = new D(); C* p4 = new E();
const A* p5 = new D(); const A* p6 = new E(); const A* p7 = new F(); F f;
```

Queste definizioni compilano correttamente (con opportuni `#include` e `using`). Per ognuno dei seguenti statement scrivere nell'apposito spazio:

- **NON COMPILA** se la compilazione dello statement provoca un errore;
- **UNDEFINED** se lo statement compila correttamente ma la sua esecuzione provoca un undefined behaviour o un errore run-time;
- se lo statement compila ed esegue correttamente (senza undefined behaviour o errori run-time) allora si scriva la stampa che l'esecuzione produce in output su `cout`; se non provoca alcuna stampa allora si scriva **NESSUNA STAMPA**.

```
(dynamic_cast<C*>(const_cast<A*>(p7)))->A::k(); .....

(dynamic_cast<E*>(p6))->h(); .....

(p5->n()).g(); .....

p4->f(); .....

p4->k(); .....

(p4->n()).m(); .....

p3->k(); .....

(dynamic_cast<D&>(p3->n())).g(); .....

(p3->n()).m(); .....

p2->m(); .....

(p2->h())->g(); .....

p1->m(); .....

(p1->h())->k(); .....

(dynamic_cast<const F*>(p1->h()))->g(); .....

C* ptr = new F(f); .....

delete p7; .....
```

Esercizio Funzione

Si assumano le seguenti specifiche (**NON È CODICE DA SCRIVERE**) di una generica libreria grafica.

A. `Component` è una classe astratta i cui oggetti, detti componenti, hanno una rappresentazione grafica che può essere mostrata sul display. La classe `Component` rende disponibile un metodo virtuale e costante `bool hasFocus()` con il seguente comportamento: una invocazione `c.hasFocus()` ritorna `true` se la componente `c` detiene il focus del display, altrimenti ritorna `false`.

B. `Container` è una sottoclasse concreta di `Component` i cui oggetti sono componenti detti contenitori che possono contenere altre componenti. La classe `Container` rende disponibile un metodo virtuale `void setHeight(double)` con il seguente comportamento: una invocazione `c.setHeight(d)` imposta l'altezza `d` in cm per il contenitore `c`. La classe `Container` rende inoltre disponibile un metodo virtuale `void setWidth(double)` con il seguente comportamento: una invocazione `c.setWidth(d)` imposta la larghezza `d` in cm per il contenitore `c`.

C. `Window` è una sottoclasse di `Container` i cui oggetti rappresentano generiche finestre. La classe `Window` rende disponibile un metodo `void hide()` con il seguente comportamento: una invocazione `w.hide()` nasconde la finestra `w` se `w` è visibile sul display, altrimenti, cioè se `w` è nascosta, lancia un oggetto eccezione di un tipo `Hidden` dotato di costruttore di default. La classe `Window` rende inoltre disponibile un metodo virtuale e costante `bool hasMenu()` con il seguente comportamento: una invocazione `w.hasMenu()` ritorna `true` se alla finestra `w` è stato impostato un menu, altrimenti ritorna `false`. La classe `Window` fornisce l'overriding dei metodi virtuali `void setHeight(double)` e `void setWidth(double)` specializzandoli per la classe `Window`.

D. `Frame` è una sottoclasse di `Window` i cui oggetti rappresentano finestre grafiche con titolo e bordo (dette frame). La classe `Frame` rende disponibile un metodo virtuale `void setTitle(string)` con il seguente comportamento: una invocazione `f.setTitle(s)` imposta alla stringa `s` il titolo del frame `f`. La classe `Frame` fornisce l'overriding dei metodi virtuali `void setHeight(double)` e `void setWidth(double)` specializzandoli per la classe `Frame`.

Definire una funzione `void fun(const Component&, vector<const Window*>&)` con il seguente comportamento: in ogni invocazione `fun(c,v)`:

- se `c` è un frame a cui è stato impostato un menu allora imposta alla stringa "`menu`" il titolo del frame `c` e inserisce un puntatore a `c` nel vettore `v`;
- se `c` è una generica finestra visibile sul display allora nasconde la finestra `c`;
- se `c` è un contenitore che detiene il focus del display allora imposta altezza e larghezza di `c` entrambe a 3cm;
- in tutti gli altri casi, esce normalmente senza provocare alcun effetto, in particolare quindi senza lanciare alcuna eccezione.

SOLUZIONE