Laurea in Informatica – Programmazione ad Oggetti – Appello d'Esame 6/2/2020

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

Esercizio 1

Scrivere un template di classe `SmartP<T>` di **puntatori smart** al parametro di tipo `T` che ridefinisca assegnazione profonda, costruzione di copia profonda e distruzione profonda di puntatori smart. Il template `SmartP<T>` dovrà essere dotato della **minima** interfaccia pubblica (cioè con il minor numero di membri) 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
  cout << (s == t) << " " << !(s == p) << endl; // 0 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
  C c; SmartP<C> x(&c); SmartP<C> y(x);
  cout << (x == y) << endl; // 0
  cout << *(c.p) << endl; // 5
  *(c.p)=6;
  cout << *(c.p) << endl; // 6
  SmartP<C>* q = new SmartP<C>(&c);
  delete q;
}
```

Esercizio 2

Si considerino le seguenti definizioni.

```
class B {
private:
  vector<bool>* ptr;
  virtual void m() const =0;
};

class D: public B {
private:
  int x;
};

class F: public D {
private:
  list<int*> l;
  int& ref;
  double* p;
public:
  void m() const {}
  // ridefinizione del costruttore di copia di F
};
```

Ridefinire il costruttore di copia della classe `F` in modo tale che il suo comportamento coincida con quello del costruttore di copia standard di `F`.

Esercizio 3

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

```
class B: public A {
public:
  virtual void g() const {cout <<" B::g ";}
  virtual void m() {cout <<" B::m "; g(); r();}
  void k() {cout <<" B::k "; A::n();}
  A* n() {cout <<" B::n "; return this;}
};
```

```
class C: public A {
protected:
  void r() {cout <<" C::r ";}
public:
  virtual void g() {cout <<" C::g ";}
  void m() {cout <<" C::m "; g(); r();}
  void k() const {cout <<" C::k "; k();}
};
```

```
class D: public B {
protected:
  void r() {cout <<" D::r ";}
public:
  B* n() {cout <<" D::n "; return this;}
  void m() {cout <<" D::m "; g(); r();}
};
```

```
A* p1 = new D(); A* p2 = new B(); A* p3 = new C(); B* p4 = new D(); const A* p5 = new C();
```

Le precedenti definizioni compilano correttamente. Per ognuna delle seguenti istruzioni scrivere nell'apposito spazio:

- **NON COMPILA** se la compilazione dell'istruzione provoca un errore;
- **ERRORE RUN-TIME** se l'istruzione compila correttamente ma la sua esecuzione provoca un errore a run-time;
- se l'istruzione compila correttamente e non provoca errori a 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**.

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

p2->f();  .....

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

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

p3->f();  .....

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

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

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

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

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

(dynamic_cast<B*>(p1))->m(); .....

(static_cast<C*>(p2))->k();  .....
```