Laurea in Informatica – Programmazione ad Oggetti – Appello d'Esame 1/2/2016

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

È VIETATO l'uso di oggetti diversi dalla penna. Scrivere le soluzioni CHIARAMENTE nel foglio a quadretti.

Esercizio 1

Si consideri il seguente modello concernente una app Gallo© che offre delle funzionalità di galleria di foto/video.

(A) Definire la seguente gerarchia di classi.

1. Definire una classe base polimorfa astratta `GalloFile` i cui oggetti rappresentano un file grafico memorizzabile dalla app Gallo©. Ogni `GalloFile` è caratterizzato dalla dimensione in MB. La classe include un metodo virtuale puro di "clonazione" `GalloFile* clone()`, che prevede il contratto standard della clonazione polimorfa di oggetti, ed un metodo virtuale puro `bool highQuality()` con il seguente contratto puro: `f->highQuality()` ritorna true se il file grafico `*f` è considerato di alta qualità, altrimenti ritorna false.
2. Definire una classe concreta `Foto` derivata da `GalloFile` i cui oggetti rappresentano un file immagine scattato da una fotocamera. Ogni oggetto `Foto` è caratterizzato dalla sensibilità ISO della fotocamera (un intero positivo) e dall'essere stata scattata con il flash oppure senza flash. La classe `Foto` implementa i metodi virtuali puri nel seguente modo: `f.clone()` ritorna un puntatore ad un oggetto `Foto` che è una copia di `f`; inoltre, `f.highQuality()` ritorna true quando la sensibilità ISO di `f` è almeno 500.
3. Definire una classe concreta `Video` derivata da `GalloFile` i cui oggetti rappresentano un file video girato da una fotocamera. Ogni oggetto `Video` è caratterizzato dalla durata in secondi e dall'essere in formato FullHD o superiore oppure no. La classe `Video` implementa i metodi virtuali puri nel seguente modo: `v.clone()` ritorna un puntatore ad un oggetto `Video` che è una copia di `v`; inoltre, `v.highQuality()` ritorna true quando `v` è in formato FullHD o superiore.

(B) Definire una classe `Gallo` i cui oggetti rappresentano una installazione dell'app Gallo©. Un oggetto `g` di `Gallo` è quindi caratterizzato dall'insieme di tutti i file grafici memorizzati da `g`. La classe `Gallo` rende disponibili i seguenti metodi:

1. Un metodo `vector<GalloFile*> selectHQ()` con il seguente comportamento: una invocazione `g.selectHQ()` ritorna il vector dei puntatori ai `GalloFile` memorizzati in `g` che: (i) sono considerati di alta qualità e (ii) se sono una `Foto` allora devono essere stati scattati con il flash.
2. Un metodo `void removeNonFoto(double)` con il seguente comportamento: una invocazione `g.removeNonFoto(size)` elimina tutti i file grafici memorizzati da `g` che: (i) non sono una `Foto` e (ii) hanno una dimensione maggiore di `size`; nel caso non venga eliminato alcun file grafico da `g` allora deve essere sollevata l'eccezione `std::logic_error("NoRemove")`, ricordando che `std::logic_error` è un tipo di eccezioni della libreria standard.
3. Un metodo `const GalloFile* insert(const GalloFile* pf)` con il seguente comportamento: se l'oggetto `*pf` non è un `Video` oppure se `*pf` è un `Video` di durata minore ad un minuto allora l'invocazione `g.insert(pf)` inserisce `*pf` tra i file grafici memorizzati da `g` e quindi ritorna un puntatore all'oggetto inserito; se invece `*pf` non viene inserito tra i file grafici memorizzati da `g` allora ritorna il puntatore nullo.

Esercizio 2

Si considerino le seguenti definizioni.

```
class B {
private:
  list<double>* ptr;
  virtual void m() =0;
};

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

class E: public C, public D {
private:
  vector<int*> v;
public:
  void m() {}
  // ridefinizione del costruttore di copia di E
};
```

```
class C: virtual public B {};
```

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

Esercizio 3

Si considerino le seguenti definizioni:

```
class A {
public:
  virtual void m() =0;
};

class B: virtual public A {};

class C: virtual public A {
public:
  virtual void m() {}
};

class D: public B, public C {
public:
  virtual void m() {}
};

class E: public D {};

class F: public E {};

char G(A* p, B& r) {
  C* pc = dynamic_cast<E*>(&r);
  if(pc && typeid(*p)==typeid(r)) return 'G';
  if(!dynamic_cast<E*>(&r) && dynamic_cast<D*>(p)) return 'Z';
  if(!dynamic_cast<F*>(pc)) return 'A';
  else if(typeid(*p)==typeid(E)) return 'S';
  return 'E';
}
```

Si consideri inoltre il seguente statement.

```
  cout << G(new X1,*new Y1) << G(new X2,*new Y2) << G(new X3,*new Y3) << G(new X4,*new Y4)
       << G(new X5,*new Y5) << G(new X6,*new Y6) << G(new X7,*new Y7) << G(new X8,*new Y8);
```

Definire opportunamente le incognite di tipo `Xi` e `Yi` tra i tipi `A`, `B`, `C`, `D`, `E`, `F` della precedente gerarchia in modo tale che:

1. Lo statement non includa piú di una chiamata della funzione `G` con gli stessi parametri attuali
2. La compilazione dello statement non produca illegalità
3. L'esecuzione dello statement non provochi errori a run-time
4. L'esecuzione dello statement produca in output esattamente la stampa **`SAGGEZZA`**.

Soluzione:

```
X1 = ............   Y1 = ............

X2 = ............   Y2 = ............

X3 = ............   Y3 = ............

X4 = ............   Y4 = ............

X5 = ............   Y5 = ............

X6 = ............   Y6 = ............

X7 = ............   Y7 = ............

X8 = ............   Y8 = ............
```