Esercizi di Programmazione ad Oggetti a.a. 19/20

prof. Francesco Ranzato

Esercizio 1

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
}
```

Esercizio 2

Si consideri il seguente modello di realtà concernente l'app InForma per archiviare allenamenti sportivi.

(A) Definire la seguente gerarchia di classi.

1. Definire una classe base polimorfa astratta `Workout` i cui oggetti rappresentano un allenamento (workout) archiviabile in InForma. Ogni `Workout` è caratterizzato dalla durata temporale espressa in minuti. La classe è astratta in quanto prevede i seguenti **metodi virtuali puri**:
   - un metodo di "clonazione": `Workout* clone()`.
   - un metodo `int calorie()` con il seguente contratto puro: `w->calorie()` ritorna il numero di calorie consumate durante l'allenamento `*w`.
2. Definire una classe concreta `Corsa` derivata da `Workout` i cui oggetti rappresentano un allenamento di corsa. Ogni oggetto `Corsa` è caratterizzato dalla distanza percorsa espressa in Km. La classe `Corsa` implementa i metodi virtuali puri di `Workout` come segue:
   - implementazione della clonazione standard per la classe `Corsa`.
   - per ogni puntatore `p` a `Corsa`, `p->calorie()` ritorna il numero di calorie dato dalla formula $500K^2/D$, dove K è la distanza percorsa in Km nell'allenamento `*p` e D è la durata in minuti dell'allenamento `*p`.
3. Definire una classe astratta `Nuoto` derivata da `Workout` i cui oggetti rappresentano un generico allenamento di nuoto che non specifica lo stile di nuoto. Ogni oggetto `Nuoto` è caratterizzato dal numero di vasche nuotate.
4. Definire una classe concreta `StileLibero` derivata da `Nuoto` i cui oggetti rappresentano un allenamento di nuoto a stile libero. La classe `StileLibero` implementa i metodi virtuali puri di `Nuoto` come segue:
   - implementazione della clonazione standard per la classe `StileLibero`.
   - per ogni puntatore `p` a `StileLibero`, `p->calorie()` ritorna il seguente numero di calorie: se D è la durata in minuti dell'allenamento `*p` e V è il numero di vasche nuotate nell'allenamento `*p` allora quando $D < 10$ le calorie sono $35V$, mentre se $D \geq 10$ le calorie sono $40V$.

5. Definire una classe concreta `Dorso` derivata da `Nuoto` i cui oggetti rappresentano un allenamento di nuoto a stile dorso. La classe `Dorso` implementa i metodi virtuali puri di `Nuoto` come segue:
   - implementazione della clonazione standard per la classe `Dorso`.
   - per ogni puntatore `p` a `Dorso`, `p->calorie()` ritorna il seguente numero di calorie: se D è la durata in minuti dell'allenamento `*p` e V è il numero di vasche nuotate nell'allenamento `*p` allora quando $D < 15$ le calorie sono $30V$, mentre se $D \geq 15$ le calorie sono $35V$.
6. Definire una classe concreta `Rana` derivata da `Nuoto` i cui oggetti rappresentano un allenamento di nuoto a stile rana. La classe `Rana` implementa i metodi virtuali puri di `Nuoto` come segue:
   - implementazione della clonazione standard per la classe `Rana`.
   - per ogni puntatore `p` a `Rana`, `p->calorie()` ritorna $25V$ calorie dove V è il numero di vasche nuotate nell'allenamento `*p`.

(B) Definire una classe `InForma` i cui oggetti rappresentano una installazione dell'app. Un oggetto di `InForma` è quindi caratterizzato da un contenitore di elementi di tipo `const Workout*` che contiene tutti gli allenamenti archiviati dall'app. La classe `InForma` rende disponibili i seguenti metodi:

1. Un metodo `vector<Nuoto*> vasche(int)` con il seguente comportamento: una invocazione `app.vasche(v)` ritorna un STL vector di puntatori a **copie** di tutti e soli gli allenamenti a nuoto memorizzati in `app` con un numero di vasche percorse $>$ `v`.
2. Un metodo `vector<Workout*> calorie(int)` con il seguente comportamento: una invocazione `app.calorie(x)` ritorna un vector contenente dei puntatori a **copie** di tutti e soli gli allenamenti memorizzati in `app` che : (i) hanno comportato un consumo di calorie $>$ `x`; e (ii) non sono allenamenti di nuoto a rana.
3. Un metodo `void removeNuoto()` con il seguente comportamento: una invocazione `app.removeNuoto()` rimuove dagli allenamenti archiviati in `app` **tutti** gli allenamenti a nuoto che abbiano il massimo numero di calorie tra tutti gli allenamenti a nuoto.

Esercizio 3

Si considerino le seguenti dichiarazioni di classi di qualche libreria grafica, dove gli oggetti delle classi `Container`, `Component`, `Button` e `MenuItem` sono chiamati, rispettivamente, contenitori, componenti, pulsanti ed entrate di menu.

```
class Component;

class Container {
public:
  virtual ~Container();
  vector<Component*> getComponents() const;
};

class Component: public Container {};

class Button: public Component {
public:
  vector<Container*> getContainers() const;
};

class MenuItem: public Button {
public:
  void setEnabled(bool b = true);
};

class NoButton {};
```

Si assumino i seguenti fatti.

1. Il comportamento del metodo `getComponents()` della classe `Container` è il seguente: `c.getComponents()` ritorna un vector di puntatori a tutte le componenti inserite nel contenitore `c`; se `c` non ha alcuna componente allora ritorna un vector vuoto.
2. Il comportamento del metodo `getContainers()` della classe `Button` è il seguente: `b.getContainers()` ritorna un vector di puntatori a tutti i contenitori che contengono il pulsante `b`; se `b` non appartiene ad alcun contenitore allora ritorna un vector vuoto.

3. Il comportamento del metodo `setEnabled()` della classe `MenuItem` è il seguente: `mi.setEnabled(b)` abilita (con `b==true`) o disabilita (con `b==false`) l'entrata di menu `mi`.

Definire una funzione `Button** Fun(const Container&)` con il seguente comportamento: in ogni invocazione `Fun(c)`

1. Se `c` contiene almeno una componente `Button` allora

   ritorna un puntatore alla prima cella di un array dinamico di puntatori a pulsanti contenente tutti e soli i puntatori ai pulsanti che sono componenti del contenitore `c`; inoltre tutte le componenti del contenitore `c` che sono una entrata di menu e sono contenute in almeno 2 contenitori devono essere disabilitate.

2. Se invece `c` non contiene nessuna componente `Button` allora ritorna il puntatore nullo.

Esercizio 4

Si assuma che `Abs` sia una classe base astratta. Definire un template di funzione `Fun(T1*, T2&)` che ritorna un booleano con il seguente comportamento. Consideriamo una istanziazione implicita `Fun(p,r)` dove supponiamo che i parametri di tipo `T1` e `T2` siano istanziati a tipi polimorfi (cioè che contengono almeno un metodo virtuale). Allora `Fun(p,r)` ritorna `true` se e soltanto se valgono le seguenti condizioni:

1. i parametri di tipo `T1` e `T2` sono istanziati allo stesso tipo;
2. siano `D1*` il tipo dinamico di `p` e `D2&` il tipo dinamico di `r`. Allora (i) `D1` e `D2` sono lo stesso tipo e (ii) questo tipo è un sottotipo proprio della classe `Abs`.

Esercizio 5

Sia `B` una classe polimorfa e sia `C` una sottoclasse di `B`. Definire una funzione `int Fun(const vector<B*>& v)` con il seguente comportamento: sia `v` non vuoto e sia `T*` il tipo dinamico di `v[0]`; allora `Fun(v)` ritorna il numero di elementi di `v` che hanno un tipo dinamico `T1*` tale che `T1` è un sottotipo di `C` diverso da `T`; se `v` è vuoto deve quindi ritornare 0. Ad esempio, il seguente programma deve compilare e provocare le stampe indicate.

```
#include<iostream>
#include<typeinfo>
#include<vector>
using namespace std;

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

int Fun(vector<B*> &v){...}

main() {
  vector<B*> u, v, w;
  cout << Fun(u); // stampa 0
  B b; C c; D d; E e; B *p = &e, *q = &c;
  v.push_back(&c); v.push_back(&b); v.push_back(&d); v.push_back(&c);
  v.push_back(&e); v.push_back(p);
  cout << Fun(v); // stampa 2
  w.push_back(p); w.push_back(&d); w.push_back(q); w.push_back(&e);
  cout << Fun(w); // stampa 1
}
```

Esercizio 6

Si consideri il seguente modello di realtà. L'operatore di telefonia mobile Moon offre ai clienti due tipologie di tariffazione: al minuto ed abbonamento. Moon offre gli usuali servizi mobili: telefonate, SMS, connessione dati. Siamo interessati al costo mensile dei servizi Moon per i clienti.

(A) Definire una classe `Cliente` i cui oggetti rappresentano un cliente Moon. Ogni `Cliente` è caratterizzato da: il tempo (in minuti) di telefonate effettuate nel mese corrente; il numero di telefonate effettuate nel mese corrente; il numero di SMS inviati nel mese corrente; il traffico dati (in MB) effettuato nel mese corrente. Per tutti i clienti Moon, il costo del traffico dati è fissato in 0.01 € per MB.

- La classe `Cliente` dichiara un metodo virtuale puro `double costoMeseCorrente()` che prevede il seguente contratto: una invocazione `c.costoMeseCorrente()` ritorna il costo attualmente da pagare per il mese corrente dal cliente `c`.

(B) Definire una classe `AlMinuto` derivata da `Cliente` i cui oggetti rappresentano un cliente Moon con tariffazione al minuto. Per tutti i clienti Moon con tariffazione `AlMinuto` sono vigenti i seguenti costi: scatto alla risposta: 0.15 €; costo delle telefonate: 0.2 € al minuto; costo di un SMS: 0.1 €.

La classe `AlMinuto` implementa `costoMeseCorrente()` nel seguente modo: una invocazione `c.costoMeseCorrente()` ritorna il costo attualmente da pagare dal cliente con tariffazione al minuto `c` ottenuto sommando gli scatti alla risposta per tutte le telefonate effettuate da `c`, il costo a tempo delle telefonate effettuate da `c`, il costo degli SMS inviati da `c`, il costo del traffico dati effettuato da `c`

(C) Definire una classe `Abbonamento` derivata da `Cliente` i cui oggetti rappresentano un cliente Moon con tariffazione ad abbonamento. Ogni cliente Moon con tariffazione `Abbonamento` è caratterizzato dal costo fisso mensile dell'abbonamento (che può quindi variare tra i clienti) e dalla soglia di tempo mensile espresso in minuti di telefonate incluse nell'abbonamento (che può pure variare tra i clienti). Per tutti i clienti `Abbonamento`, il costo delle telefonate oltre la soglia di tempo mensile è fissato in 0.3 € al minuto.

La classe `Abbonamento` implementa `costoMeseCorrente()` nel seguente modo: una invocazione `c.costoMeseCorrente()` ritorna il costo attualmente da pagare dal cliente con tariffazione ad abbonamento `c` dato dal costo fisso mensile per `c` piú il costo del traffico dati effettuato da `c` piú l'eventuale costo delle telefonate effettuate oltre la soglia di tempo mensile per `c`. Quindi, la tariffazione ad abbonamento include gli SMS illimitati.

(D) Definire una classe `sedeMoon` i cui oggetti rappresentano l'insieme dei clienti gestiti da una sede Moon. Devono essere disponibili nella classe `sedeMoon` le seguenti funzionalità:

- Un metodo `void aggiungiCliente(const Cliente&)` con il seguente comportamento: una invocazione `s.aggiungiCliente(c)` aggiunge `c` all'insieme dei clienti gestiti dalla sede Moon `s`.
- Un metodo `double incassoMensileCorrente()` con il seguente comportamento: una invocazione `s.incassoMensileCorrente()` ritorna l'incasso mensile corrente della sede Moon `s` da tutti i clienti gestiti da `s`.
- Un metodo `int numClientiAbbonatiOltreSoglia()` con il seguente comportamento: una invocazione `s.numClientiAbbonatiOltreSoglia()` ritorna il numero di clienti con tariffazione ad abbonamento della sede Moon `s` che hanno superato la loro soglia di tempo mensile di telefonate.
- Un metodo `vector<AlMinuto> alMinutoSMS(int)` con il seguente comportamento: una invocazione `s.alMinutoSMS(x)` ritorna un vettore contenente una copia di tutti e soli i clienti con tariffazione al minuto gestiti dalla sede Moon `s` che hanno inviato un numero di SMS superiore a `x`.

Esercizio 7

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`.

Esercizio 8

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

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

class C: public B {
protected:
  virtual void h() const {cout <<" C::h ";}
public:
  virtual void g() {cout <<" C::g ";}
  void m() {cout <<" C::m "; g(); k();}
  void k() const {cout <<" C::k "; h();}
};

A* p2 = new B(); A* p3 = new C(); B* p4 = new B(); B* p5 = new C(); const A* p6 = 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**.

```
p2->f(); ..........
p2->m(); ..........
p3->k(); ..........
p3->f(); ..........
p4->m(); ..........
p4->k(); ..........
p4->g(); ..........
p5->g(); ..........
p6->k(); ..........
p6->g(); ..........
(p3->n())->m(); ..........
(p3->n())->g(); ..........
(p3->n())->n()->g(); ..........
(p5->n())->g(); ..........
(p5->n())->m(); ..........
(dynamic_cast<B*>(p2))->m(); ..........
(static_cast<C*>(p3))->k(); ..........
( static_cast<B*>(p3->n()) )->g(); ..........
```