

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

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

Esercizio 1 (NB: Per comodità di correzione, definire tutti i metodi inline)

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; se `app` non ha archiviato alcun allenamento a nuoto allora viene sollevata l’eccezione “NoRemove” di tipo `std::string`.

Esercizio 2

```
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();