Tutorato 7

20/12/2023

Programmazione ad Oggetti – 2023-2024

Gabriel Rovesti

2103389 – LM Computer Science

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO MATEMATICA

First thing first…

Registrazione presenza Tutorato PaO

Login con SSO su Google Form inserendo i propri dati

Esercizio 1: Modellazione

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 $40V$.

Esercizio 1: Modellazione

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: Ridefinizioni

Esercizio Definizioni

```
class Z {
private:
  int x;
};

class B {
private:
  Z bz;
};

class C: virtual public B {
private:
  Z cz;
};

class D: public C {
};

class E: virtual public B {
public:
  Z ez;
  // ridefinizione assegnazione
  // standard di E
};

class F: public D, public E {
private:
  Z* fz;
public:
  // ridefinizione del costruttore di copia profonda di F
  // ridefinizione del distruttore profondo di F
  // definizione del metodo di clonazione di F
};
```

Si considerino le definizioni sopra.
(1) Ridefinire l'assegnazione della classe `E` in modo tale che il suo comportamento coincida con quello dell'assegnazione standard di `E`. Naturalmente non è permesso l'uso della keyword `default`.
(2) Ridefinire il costruttore di copia profonda della classe `F`.
(3) Ridefinire il distruttore profondo della classe `F`.
(4) Definire il metodo di clonazione della classe `F`.

Esercizio 2: Soluzione


```
// SOLUZIONE
class E: virtual public B {
public:
  Z ez;
  E& operator(const E& e) {
    B::operator=(e);
    ez=e.ez;
    return *this;
  }
};

class F: public D, public E {
private:
  Z* fz;
public:
  F(const F& f): B(f), D(f), E(f), fz(f.fz!=nullptr ? new Z(*f.fz) : nullptr) {}
  ~F() {delete fz;}
  virtual F* clone() const {return new F(*this);}
};
```

Esercizio 3: Sottotipi

Esercizio Tipi

```
class A {
public:
  virtual ~A() = 0;
};
A::~A() = default;

class B: public A {
public:
  ~B() = default;
};

char F(const A& x, B* y) {
  B* p = const_cast<B*>(dynamic_cast<const B*> (&x));
  auto q = dynamic_cast<const C*> (&x);
  if(dynamic_cast<E*> (y)) {
    if(!p || q) return '1';
    else return '2';
  }
  if(dynamic_cast<C*> (y)) return '3';
  if(q) return '4';
  if(p && typeid(*p) != typeid(D)) return '5';
  return '6';
}
```

```
class C: virtual public B {};
class D: virtual public B {};
class E: public C, public D {};
```

```
int main() {
B b; C c; D d; E e;


cout << F(...,...) << F(...,...) << F(...,...) << F(...,...) << F(...,...)


     << F(...,...) << F(...,...) << F(...,...) << F(...,...) << F(...,...);
}
```

Si considerino le precedenti definizioni ed il `main()` incompleto. Definire opportunamente negli appositi spazi `...,...` le chiamate alla funzione `F` di questo `main()` usando gli oggetti locali `b`, `c`, `d`, `e`, `f` in modo tale che: (1) non vi siano errori in compilazione o a run-time; (2) le chiamate di `F` siano **tutte diverse** tra loro; (3) l'esecuzione produca in output **esattamente** la stampa **`6544233241`**.

Esercizio 3: Sottotipi


```
class A {
public:
  virtual ~A() = 0;
};
A::~A() = default;

class B: public A {
public:
  ~B() = default;
};

char F(const A& x, B* y) {
  B* p = const_cast<B*>(dynamic_cast<const B*> (&x));
  auto q = dynamic_cast<const C*> (&x);
  if(dynamic_cast<E*> (y)) {
    if(!p || q) return '1';
    else return '2';
  }
  if(dynamic_cast<C*> (y)) return '3';
  if(q) return '4';
  if(p && typeid(*p) != typeid(D)) return '5';
  return '6';
}
```

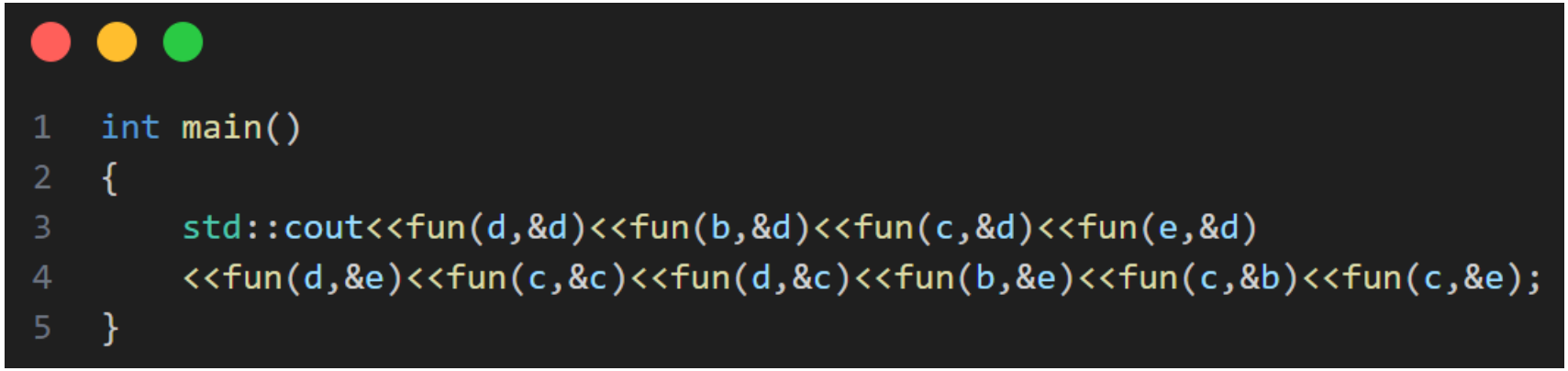
```
class C: virtual public B {};
class D: virtual public B {};
class E: public C, public D {};
```

```
int main() {
B b; C c; D d; E e;


cout << F(....,....) << F(....,....) << F(....,....) << F(....,....) << F(....,....)


     << F(....,....) << F(....,....) << F(....,....) << F(....,....) << F(....,....);
}
```

Esercizio 3: Soluzione (possibile)



```
1  int main()
2  {
3      std::cout<<fun(d,&d)<<fun(b,&d)<<fun(c,&d)<<fun(e,&d)
4      <<fun(d,&e)<<fun(c,&c)<<fun(d,&c)<<fun(b,&e)<<fun(c,&b)<<fun(c,&e);
5  }
```