Esercizi di Programmazione ad Oggetti

Lista n. 6

Esercizio 1

```
class B {
public:
  int b;
  explicit B(int x=1): b(x) {}
  virtual B* m(B& x) {return new B(b + x.b);}
  virtual void print() {cout << b << " ";}
};
class C: public B {
public:
  int c;
  explicit C(int x=2): B(x), c(x) {}
  void print() {B::print(); cout << c << " ";}
  void f() {B* x = m(*this); x->print();}
};
class D: public C {
public:
  int d;
  explicit D(int x=3): C(x), d(x) {}
  B* m(B& x) {
    C* p = dynamic_cast<C*>(&x);
    D* q = dynamic_cast<D*>(&x);
    if(!p) return new C(d + x.b);
    if(q) return new D(d + q->d);
    return new B(x.b);
  }
  void print() {C::print(); cout << d << " ";}
};

main(){
  B b(1); C c; D d;
  B* p1 = new D(3); B* x;
  B* p2 = p1->m(*p1); p2->print(); cout << " **1\n";
  x = p1->m(c); x->print(); cout << " **2\n";
  x = p1->m(b); x->print(); cout << " **3\n";
  x = p2->m(*p1); x->print(); cout << " **4\n";
  x = x->m(b); x->print(); cout << " **5\n";
  C* p3 = new C(4); p3->f(); cout << " **6\n";
  p3 = &d; p3->f(); cout << " **7\n";
  (dynamic_cast<C*>(p3->m(d)))->f(); cout << " **8";
}
```

Le precedenti definizioni compilano ed eseguono correttamente. Quali stampe provoca in output l'esecuzione del `main()`?

Esercizio 2

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 3

Il seguente programma compila. Cosa stampa in output? Si ricordi che dati due iteratori `first` e `last` su un vector `v` la funzione `find(first,last,value)` ritorna il primo iteratore `it` nell'intervallo `[first,last)` tale che `*it==value`, mentre se tale iteratore non esiste ritorna `last`.

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

class B {
public:
  int k;
  B(int x=1): k(x) {}
  virtual ~B() {}
};

class C: public B {
public:
  C(): B(2) {}
};

class D {
public:
  B* punt;

  D(B* p): punt(p) {
    if(typeid(*p)==typeid(B)) punt = new B(p->k);
  }
  D(const D& d): punt(d.punt) {
    if(typeid(*(d.punt))==typeid(B)) punt = new B((d.punt)->k);
  }
  D& operator=(const D& d) {
    if(this != &d) {
      if((typeid(*(d.punt))==typeid(B))) punt = new B((d.punt)->k);
      else punt = d.punt;
    }
    return *this;
  }
};

main(){
  B b1(4), b2; C c1;
  D d1(&c1), d2(&b1), d3(&b2), d4(d1), d5(d2); d5=d1;
  vector<D> v; vector<B*> w;
  v.push_back(d1); v.push_back(d2); v.push_back(d3);
  v.push_back(d3); v.push_back(d4); v.push_back(d5);
  for(int i=0; i < v.size(); i++)
    if(find(w.begin(),w.end(),v[i].punt)==w.end()) w.push_back(v[i].punt);
  for(int i=0; i< w.size(); i++)
```

```
    cout << w[i]->k << ' ';
}
```

Quali stampe produce la sua esecuzione?

Esercizio 4

Definire una superclasse `Biglietto` i cui oggetti rappresentano generici biglietti d'ingresso per uno spettacolo (ad esempio un concerto) e due sue sottoclassi `PostoNumerato` e `PostoNonNumerato`, i cui oggetti rappresentano, rispettivamente, biglietti per posti numerati e non numerati (naturalmente ogni biglietto sarà per un posto numerato o non numerato). Ci interesserà il costo di tali biglietti per un certo spettacolo. Queste classi devono soddisfare le seguenti specifiche:

- Ogni biglietto è caratterizzato dal nome dell'acquirente. I posti sono suddivisi tra platea e galleria. Tutti i posti numerati sono in platea, mentre i posti non numerati possono essere sia in platea che in galleria (quindi in platea vi possono essere sia posti numerati che non numerati). La classe `Biglietto` deve soddisfare la seguente condizione: Non deve essere possibile costruire oggetti di `Biglietto` in una classe che non derivi da `Biglietto`.
- Ogni biglietto per un posto numerato (quindi solo di platea) è caratterizzato dalla fila del posto.
- Come già detto, ogni biglietto per un posto non numerato può essere di galleria o di platea. Inoltre, un biglietto per un posto non numerato può essere un biglietto a prezzo ridotto o pieno.

Definire inoltre una classe `Spettacolo` i cui oggetti rappresentano uno spettacolo. La classe `Spettacolo` deve soddisfare le seguenti specifiche.

- Ogni spettacolo è caratterizzato da una base di prezzo B per i biglietti, da una addizionale di prezzo A, dal numero massimo di posti numerati, dal numero di fila n che caratterizza i posti numerati di prima fila, cioè tale che ogni posto numerato con un numero di fila $\leq n$ è un posto numerato di prima fila, ed infine dal numero di posti numerati venduti. Inoltre, ogni spettacolo è caratterizzato da una `list` di puntatori a biglietti, cioè un oggetto `list<Biglietto*>`, che rappresenta la lista dei biglietti venduti.
- La classe `Spettacolo` fornisce un metodo `aggiungiBiglietto(Biglietto& b)` che aggiunge il biglietto `b` alla lista dei biglietti venduti secondo il seguente comportamento. Se `b` è un biglietto per un posto non numerato allora `b` viene sempre aggiunto alla lista (si suppone che i posti non numerati siano illimitati). Se `b` è un biglietto per un posto numerato allora `b` viene aggiunto alla lista se vi sono ancora posti numerati disponibili (altrimenti non viene aggiunto alla lista dei biglietti venduti). In tal caso, naturalmente, viene anche aggiornato il numero di posti numerati venduti.
- La classe `Spettacolo` fornisce un metodo `prezzo(const Biglietto& b)` che calcola il prezzo del biglietto `b` come segue:
  - un biglietto per un posto numerato di prima fila costa $2 * A + 2 * B$ mentre un posto numerato non di prima fila costa $2 * A$;
  - un biglietto per un posto non numerato di galleria costa B, di platea $B + A$, e se si tratta di un biglietto a prezzo ridotto in entrambi i casi viene detratto $A/2$.
- Infine, la classe `Spettacolo` fornisce un metodo `incasso()` che ritorna l'incasso per i biglietti finora venduti (cioè quelli memorizzati nella lista dei biglietti venduti).

Definire infine un esempio di `main()` in cui viene costruito un oggetto `s` di `Spettacolo` che specifica i parametri di uno spettacolo, vengono costruiti quattro biglietti che vengono aggiunti allo spettacolo `s` e viene quindi calcolato e stampato l'incasso per quei biglietti.

Esercizio 5

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

```
class B: public A {
private:
  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(); m();}
  B* n() {cout <<" B::n "; return this;}
};

B* b = new B(); A* a = new B();
```

La compilazione delle precedenti definizioni non provoca errori (con gli opportuni `include` e `using`). Si supponga che ognuno dei seguenti frammenti sia il codice di un `main()` che può accedere alle precedenti definizioni. Tali `main()` compilano o provocano un errore run-time? In caso compilino ed eseguano senza errori, quali stampe provocano in output?

```
b->f();
```

```
b->m();
```

```
b->k();
```

```
a->f();
```

```
a->m();
```

```
a->k();
```

```
(b->n())->g();
```

```
(b->n())->n()->g();
```

```
(a->n())->g();
```

```
(a->n())->m();
```

Esercizio 6

```
class B {
public:
  int x;
  B(int z=1): x(z) {}
};

class D: public B {
public:
  int y;
  D(int z=5): B(z-2), y(z) {}
};

void fun(B* a, int size) {
  for(int i=0; i<size; ++i) cout << (*(a+i)).x << " ";
}

int main() {
  fun(new D[4],4); cout << "**1\n";
  B* b = new D[4]; fun(b,4); cout << "**2\n";
  b[0] = D(6); b[1] = D(9);  fun(b,4); cout << "**3\n";
  b = new B[4]; b[0] = D(6); b[1] = D(9);
  fun(b,4); cout << "**4\n";
}
```

La compilazione delle precedenti definizioni non provoca errori (con gli opportuni `include` e `using`) e la loro esecuzione non provoca errori a run-time. Si scrivano le stampe provocate in output dall'esecuzione del `main()`.

Esercizio 7

Definire un template di classe `albero<T>` i cui oggetti rappresentano un **albero 3-ario** ove i nodi memorizzano dei valori di tipo `T` ed hanno 3 figli (invece dei 2 figli di un usuale albero binario). Il template `albero<T>` deve soddisfare i seguenti vincoli:

1. Deve essere disponibile un costruttore di default che costruisce l'albero vuoto.
2. Gestione della memoria senza condivisione.
3. Overloading dell'operatore di uguaglianza.
4. Overloading dell'operatore di output.