Esercizi di Programmazione ad Oggetti

Foglio VII, a.a. 2008/2009

PROF. FRANCESCO RANZATO

Esercizio 1

Il seguente programma compila ed esegue correttamente. Quali stampe produce?

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

class A {
protected:
  int k;
  A(int x=1): k(x) {}
  virtual ~A() {}
public:
  virtual void m() {cout << k << " A::m() ";}
  virtual void m(int x) {k=x; cout << k << " A::m(int) ";}
};

class B: virtual public A {
public:
  virtual void m(double y) {cout << k << " B::m(double) ";}
  virtual void m(int x) {cout << k << " B::m(int) ";}
};

class C: virtual public A {
public:
  C(int x = 2): A(x) {}
  virtual void m(int x) {cout << "C::m(int) ";}
};

class D: public B {
public:
  D(int x=3): A(x) {}
  virtual void m(double y) {cout << "D::m(double) ";}
  virtual void m() {cout << "D::m() ";}
};
class E: public D, public C {
public:
  E(int x=4): C(x) {}
  virtual void m() {cout << "E::m() ";}
  virtual void m(int x) {cout << "E::m(int) ";}
  virtual void m(double y) {cout << "E::m(double) ";}
};

main() {
  A* a[7] = {new B(),new C(),new D(),new E()};
  for(int i=0; i<4; ++i) {
    a[i]->m();
    a[i]->m(i);
    a[i]->m(3.14);
    cout << " *** " << i << endl;
  }
}
```

Esercizio 2

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

class A {
private:
```

```
  int k;
public:
  A(int x=9): k(x) {cout << "A01 ";}
  virtual ~A() {cout << k << " ~A() ";}
  virtual void m() {cout << k << " A::m() ";}
};

class B: virtual public A {
private:
  string s;
public:
  ~B() {cout << "~B() ";}
  B(string _s = "pippo"): s(_s) {cout << "B01 ";}
  virtual void m(int x) {cout << s << " B::m(int) ";}
};

class C: virtual public A {
public:
  C(int x): A(x) {}
};

class D: public B, public C {
public:
  D(int x=8): A(x), C(x) {cout << "D01 ";}
  virtual void m() {cout << "D::m() ";}
  virtual void m(int x) {cout << "D::m(int) ";}
};

class E: public D {
private:
  A a;
public:
  E(): D(5) {cout << "E() ";}
  E(const A& _a): a(_a) {cout << "E(A) ";}
  virtual void m() {cout << "E::m() ";}
  virtual void m(int x) {cout << "E::m(int) ";}
};
main() {
  B b("zagor"); cout << "ZERO\n";
  D* pd = new D(6); cout << "UNO\n";
  b = *pd; b.m(5); cout << "DUE\n";
  E* pe = new E(); cout << "TRE\n";
  E e2(b); cout << "QUATTRO\n";
  delete pd; cout << "CINQUE\n";
  pd = pe; pd->m(); cout << "SEI\n";
  E* q = dynamic_cast<E*>(pd); q->D::m(4); cout << "SETTE\n";
  delete pe; cout << "OTTO\n";
}
```

Questo programma compila correttamente. Quali stampe produce la sua esecuzione?

Esercizio 3

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 `ios` della gerarchia di classi di I/O (si ricordi che `ios` è la classe base astratta della gerarchia).

Ad esempio, il seguente `main()` deve compilare e provocare le stampe indicate:

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

class C { public: virtual ~C() {} };

main() {
  ifstream f("pippo"); fstream g("pluto"), h("zagor"); iostream* p = &h;
  C c1,c2;
  cout << Fun(&cout,cin) << endl; // stampa: 0
  cout << Fun(&cout,cerr) << endl; // stampa: 1
  cout << Fun(p,h) << endl; // stampa: 0
  cout << Fun(&f,*p) << endl; // stampa: 0
  cout << Fun(&g,h) << endl; // stampa: 1
  cout << Fun(&c1,c2) << endl; // stampa: 0
}
```

Esercizio 4

Si consideri la gerarchia di classi per l'I/O. La classe base `ios` ha il distruttore virtuale, il costruttore di copia privato ed un unico costruttore (a 2 parametri con valori di default) protetto. Diciamo che le classi derivate da `istream` ma non da `ostream` (ad esempio `ifstream`), e `istream` stessa, sono *classi di input*, le classi derivate da `ostream` ma non da `istream` (ad esempio `ofstream`), ed `ostream` stessa, sono *classi di output*, mentre le classi derivate sia da `istream` che da `ostream` sono *classi di I/O* (esempi: `iostream` e `fstream`). Quindi ogni classe di input, output o I/O è una sottoclasse di `ios`. Definire una funzione `int F(ios& ref)` che restituisce -1 se il tipo dinamico di `ref` è un riferimento ad una classe di input, 1 se il tipo dinamico di `ref` è un riferimento ad una classe di output, 0 se il tipo dinamico di `ref` è un riferimento ad una classe di I/O, mentre in tutti gli altri casi ritorna 9.

Quindi, ad esempio, il seguente `main()` provoca la stampa riportata.

```
class D : public ios {
};

main() {
  istream& b = cin;
  ostream& c = cout;
  stringstream d;
  ifstream e("pippo");
  ofstream f("pluto");
  D g;
  cout << F(b) << ' ' << F(c) << ' ' << F(d) << ' ' << F(e) << ' '
       << F(f) << ' ' << F(g) <<  endl;
  // stampa: -1 1 0 -1 1 9
}
```

Esercizio 5

Ricordiamo che nella gerarchia di classi per l'I/O la classe base astratta `ios` ha il distruttore virtuale. Si definisca una classe `C` che soddisfa le seguenti specifiche.

1. Un oggetto della classe `C` è caratterizzato da un vector di puntatori a `ios`, cioè da un oggetto di tipo `vector<ios*>`, e dal numero massimo di puntatori che questo vector può contenere. Deve essere disponibile un costruttore ad un argomento intero k, con un valore di default positivo, che determina il numero massimo k di puntatori che il vector può contenere.

2. Deve essere disponibile un metodo `insert(ios& s)` che inserisce nel vector un puntatore all'oggetto `s` quando valgono entrambe le seguenti condizioni (altrimenti lascia inalterato il vector):

   (a) il vector può contenere ancora elementi (rispetto al numero massimo possibile);

   (b) se `D&` è il tipo dinamico di `s` allora il tipo `D` è diverso da `fstream` e `stringstream`.

3. Deve essere disponibile un template di metodo `conta(T& t)`, dove `T` è un parametro di tipo, che ritorna il numero di puntatori del vector che hanno un tipo dinamico `D*` tale che il tipo `D` è un sottotipo di oppure uguale a `T`.

Ad esempio, il seguente `main()` deve compilare e provocare le stampe indicate:

```
main() {
  ifstream f("pippo"); ofstream g("mandrake");
  fstream h("pluto"), i("zagor");
  ostream* p = &g;
  stringstream s;
  C c(10);
  c.insert(f); c.insert(g); c.insert(h); c.insert(i); c.insert(*p); c.insert(s);
  istream& r=f;
  cout << c.conta(r); // stampa: 1 (e` il puntatore all'oggetto f)
}
```

Esercizio 6

Si assuma che `A`, `B`, `C`, `D` siano quattro classi polimorfe. Si consideri il seguente `main()`.

```
main() {
 A a; B b; C c; D d;
 cout << (dynamic_cast<D*>(&c) ? "0 " : "1 ");
 cout << (dynamic_cast<B*>(&c) ? "2 " : "3 ");
 cout << (!(dynamic_cast<C*>(&b)) ? "4 " : "5 ");
 cout << (dynamic_cast<B*>(&a) || dynamic_cast<C*>(&a)  ? "6 " : "7 ");
 cout << (dynamic_cast<D*>(&b) ? "8 " : "9 ");
}
```

Si supponga che tale `main()` compili ed esegua correttamente. Disegnare i diagrammi di **tutte** le possibili gerarchie per le classi `A`, `B`, `C`, `D` tali che l'esecuzione del `main()` provochi la stampa: `0 3 4 6 8`.

Esercizio 7

Il seguente programma compila. Quali stampe provoca la sua esecuzione?

```
#include<iostream>
#include<string>
using namespace std;

class B: public string {
  friend class E;
protected:
  static int i;
public:
  B() {i++; cout << i << " B() ";};
  B(string s): string(s) {i++; cout << i << " " << s << " B(string) ";};
};
int B::i=0;

class C: virtual public B {
public:
  C(int x = i) {cout << x << " C(0-1) ";};
};

class D: virtual public B {
public:
  D(): B("pluto") {cout << "D() ";};
};

class E {
public:
  D d;
  E() {(B::i)++; cout << "E() ";};
  E(D d) {(B::i)++; cout << "E(D) ";};
};
```

```
template<class T>
class F: public C, public D, public E {
public:
  T t;
  F(T x): E(D()), C() {cout << i << " F(0-1) ";};
};

main() {
  E e; cout << " UNO\n";
  F<B> f(e.d); cout << " DUE\n";
  F<B>& rf = f; cout << " TRE\n";
  F<string>* pf = new F<string>("paperino"); cout << " QUATTRO\n";
}
```

Esercizio 8

Definire una superclasse `Studente` e due sue sottoclassi `StudenteIC` e `StudenteFC` che formano una gerarchia di classi i cui oggetti rappresentano studenti di una certa Università, distinti tra studenti in corso (`StudenteIC`) e studenti fuori corso (`StudenteFC`). Ci interesserà rappresentare delle informazioni utili per il calcolo delle tasse universitarie. La gerarchia deve soddisfare le seguenti specifiche:

- Un oggetto `Studente` è caratterizzato dal nome, dal corso di laurea frequentato, dalla durata legale in anni del corso di laurea (quindi ≥ 3 e ≤ 6), dal numero totale di esami previsti dal corso di laurea (diciamo ≥ 15 e ≤ 60), dal numero di esami sostenuti (quindi non negativo e minore o uguale al numero totale di esami previsti), dal voto medio degli esami sostenuti. Tutte queste informazioni devono essere private. La classe non deve essere astratta, ma comunque deve essere progettata in modo tale che in ogni funzione esterna ed in ogni classe non derivata da essa, non sia possibile costruire oggetti di `Studente`.
- Un oggetto della sottoclasse `StudenteIC` è caratterizzato dall'anno di corso, che deve quindi essere compreso tra 1 e la durata legale in anni del corso di laurea, e dal reddito annuale del proprio nucleo familiare. Queste informazioni devono essere private. È definito un metodo pubblico `int classeDiReddito()` che ritorna la classe di reddito di uno studente in corso: classe 0 se il reddito annuale è ≤ 15000 euro e lo studente frequenta l'ultimo anno di corso, classe 1 se il reddito annuale è ≤ 15000 euro ma lo studente non frequenta l'ultimo anno di corso, classe 2 se il reddito annuale è > 15000 e ≤ 30000 euro, classe 3 se il reddito annuale è > 30000 euro.
- Un oggetto della sottoclasse `StudenteFC` è caratterizzato dal numero di anni di fuori corso, ovvero un intero ≥ 1. Tale informazione deve essere privata. È definito un metodo pubblico `bool bonus()` che determina se lo studente fuori corso ha diritto ad un bonus nella tassazione: il metodo ritorna `true` se e soltanto se il numero di esami ancora da sostenere è < 5.

Si chiede inoltre di definire esternamente alla gerarchia (quindi senza alcuna relazione di ereditarietà con classi della gerarchia) una classe `Tasse` da usarsi per determinare la tassa di iscrizione annuale per un qualsiasi studente. La classe deve rappresentare le seguenti informazioni: (1) l'importo base di tassazione annuale, (2) l'importo della penale di tassazione e (3) l'importo del bonus di tassazione. Tali informazioni non devono essere pubbliche. **La classe `Tasse` non deve essere dichiarata friend in nessun'altra classe.** La classe `Tasse` contiene un metodo pubblico statico `int calcolaTasse(Studente& s)` che calcola la tassa di iscrizione annuale dovuta dallo studente `s` nel seguente modo:

- se `s` è uno studente in corso, allora la tassa dovuta è data dall'importo base di tassazione annuale sommato con l'importo della penale di tassazione moltiplicato per la classe di reddito dello studente, e da tale somma si detrae l'importo del bonus di tassazione qualora il voto medio degli esami sostenuti è ≥ 28.
- se `s` è uno studente fuori corso, allora la tassa dovuta è data dall'importo base di tassazione annuale sommato con il triplo dell'importo della penale di tassazione moltiplicato per il numero di anni di fuori corso, e da tale somma si detrae l'importo del bonus di tassazione quando lo studente ha diritto al bonus.

Definire infine un esempio di metodo `main()` che invoca esattamente tre volte il metodo `calcolaTasse()` di `Tasse` producendo precisamente il seguente output:

```
Lo studente fuori corso di Matematica Pippo deve pagare 2000 euro di tasse.
Lo studente in corso di Informatica Pluto deve pagare 1600 euro di tasse.
Lo studente fuori corso di Fisica Paperino deve pagare 1800 euro di tasse.
```

Esercizio 9

Si considerino i seguenti fatti concernenti la libreria di I/O standard.

- Si ricorda che `ios` è la classe base di tutta la gerarchia di classi della libreria di I/O, che la classe `istream` è derivata direttamente e virtualmente da `ios` e che la classe `ifstream` è derivata direttamente da `istream`.
- La classe base `ios` ha il distruttore virtuale. La classe `ios` rende disponibile un metodo costante e non virtuale `bool fail()` con il seguente comportamento: una invocazione `s.fail()` ritorna `true` se e solo se lo stream `s` è in uno stato di fallimento (cioè, il failbit di `s` vale 1).
- La classe `istream` rende disponibile un metodo non costante e non virtuale `long tellg()` con il seguente comportamento: una invocazione `s.tellg()`:
  1. se `s` è in uno stato di fallimento allora ritorna -1;
  2. altrimenti, cioè se `s` non è in uno stato di fallimento, ritorna la posizione della cella corrente di input di `s`.
- La classe `ifstream` rende disponibile un metodo non costante e non virtuale `bool is_open()` con il seguente comportamento: una invocazione `s.is_open()` ritorna `true` se e solo se il file associato allo stream `s` è aperto.

Definire una funzione `long Fun(const ios&)` con il seguente comportamento: una invocazione `Fun(s)`:

(1) se `s` è in uno stato di fallimento lancia una eccezione di tipo `Fallimento`; si chiede anche di definire tale classe `Fallimento`;

(2) se `s` non è in uno stato di fallimento allora:

   (a) se `s` non è un `ifstream` ritorna -2;

   (b) se `s` è un `ifstream` ed il file associato non è aperto ritorna -1;

   (c) se `s` è un `ifstream` ed il file associato è aperto ritorna la posizione della cella corrente di input di `s`.

Esercizio 10

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

class A {
  friend class C;
private:
  int k;
public:
  A(int x=2): k(x) {}
  void m(int x=3) {k=x;}
};

class C {
private:
  A* p;
  int n;
public:
  C(int k=3) {if (k>0) {p = new A[k]; n=k;}}
  A* operator->() const {return p;}
  A& operator*() const {return *p;}
  A* operator+(int i) const {return p+i;}
  void F(int k, int x) {if (k<n) p[k].m(x);}
  void stampa() const {for(int i=0; i<n; i++) cout << p[i].k << ' ';}
};

main() {
  C c1;  c1.F(2,9);
  C c2(4); c2.F(0,8);
  *c1=*c2;
  (c2+3)->m(7);
  c1.stampa(); cout << "UNO\n";
  c2.stampa(); cout << "DUE\n";
```

```
  c1=c2;
  *(c2+1)=A(3);
  c1->m(1);
  *(c2+2)=*c1;
  c1.stampa(); cout << "TRE\n";
  c2.stampa(); cout << "QUATTRO";
}
```

Questo programma compila correttamente. Quali stampe produce la sua esecuzione?

Esercizio 11

Si consideri la seguente realtà concernente i biglietti del treno. Come ben noto, un biglietto per un viaggio in treno può essere di prima o seconda classe.

1. Definire una classe `Biglietto` i cui oggetti rappresentano un biglietto per un viaggio in treno. Ogni `Biglietto` è caratterizzato dalla distanza chilometrica del viaggio. La classe `Biglietto` dichiara un metodo virtuale puro `double prezzo()` che prevede il seguente contratto: una invocazione `b.prezzo()` ritorna il prezzo del biglietto `b`. Per tutti i biglietti, il prezzo base al km è fissato in 0.1 €.

2. Definire una classe `BigliettoPrimaClasse` derivata da `Biglietto` i cui oggetti rappresentano un biglietto per un viaggio di prima classe. Il prezzo di un biglietto di prima classe con distanza inferiore a 100 km è dato dal prezzo base (prezzo base al km moltiplicato per la distanza chilometrica) aumentato del 30%, altrimenti l'aumento del prezzo base è del 20%. `BigliettoPrimaClasse` implementa quindi `prezzo()` ritornando il prezzo di un dato biglietto di prima classe.

3. Definire una classe `BigliettoSecondaClasse` derivata da `Biglietto` i cui oggetti rappresentano un biglietto per un viaggio di seconda classe. Un biglietto di seconda classe può essere con prenotazione oppure senza (la prenotazione garantisce il posto a sedere). Per tutti i biglietti di seconda classe, il costo della prenotazione è fissato in 5 €. Il prezzo di un biglietto di seconda classe è dato dal prezzo base (prezzo base al km moltiplicato per la distanza chilometrica) più l'eventuale costo della prenotazione.

4. Definire una classe `BigliettoSmart` i cui oggetti rappresentano dei puntatori smart a `Biglietto`. La classe `BigliettoSmart` dovrà essere dotata dell'interfaccia pubblica necessaria per lo sviluppo della successiva classe `Treno`.

5. Definire una classe `TrenoPieno` i cui oggetti rappresentano delle eccezioni che segnalano che non vi sono posti disponibili in un dato treno. Una eccezione di `TrenoPieno` è caratterizzata dalla classe (1ˆ o 2ˆ) in cui non vi sono più posti disponibili.

6. Definire una classe `Treno` i cui oggetti rappresentano un certo viaggio in treno (la semplificazione prevede che non vi siano fermate intermedie). Ogni oggetto `Treno` è quindi caratterizzato dall'insieme dei biglietti venduti per quel viaggio in treno, e tale insieme deve essere rappresentato mediante un vector `venduti` di puntatori smart `BigliettoSmart`. Un oggetto `Treno` è inoltre caratterizzato dal numero massimo di posti disponibili per biglietti di prima classe e dal numero massimo di posti disponibili per biglietti di seconda classe con prenotazione.

   Devono essere disponibili nella classe `Treno` le seguenti funzionalità:

   - Un metodo `int* bigliettiVenduti()` con il seguente comportamento: una invocazione `t.bigliettiVenduti()` ritorna un array `ar` di 3 interi tale che:
     - `ar[0]` memorizza il numero di biglietti venduti di prima classe per il treno `t`;
     - `ar[1]` memorizza il numero di biglietti venduti di seconda classe con prenotazione per il treno `t`;
     - `ar[2]` memorizza il numero di biglietti venduti di seconda classe senza prenotazione per il treno `t`.
   - Un metodo `void vendiBiglietto(const Biglietto&)` con il seguente comportamento: una chiamata `t.vendiBiglietto(b)` aggiunge `b` tra i biglietti venduti per il treno `t` quando possibile, altrimenti solleva una opportuna eccezione di `TrenoPieno`. Più dettagliatamente:
     - Se `b` è un biglietto di prima classe e vi sono ancora posti di prima classe disponibili in `t` allora viene aggiunto al vector `venduti` un puntatore smart a `b`; se invece non vi sono posti di prima classe disponibili viene sollevata una eccezione `TrenoPieno` in prima classe.
     - Se `b` è un biglietto di seconda classe con prenotazione e vi sono ancora posti di seconda classe con prenotazione disponibili in `t` allora viene aggiunto al vector `venduti` un puntatore smart a `b`; se invece non vi sono posti di seconda classe con prenotazione disponibili viene sollevata una eccezione `TrenoPieno` in seconda classe.

- Se `b` è un biglietto di seconda classe senza prenotazione allora viene sempre aggiunto al vector `venduti` un puntatore smart a `b`.

- Un metodo `double incasso()` con il seguente comportamento: una chiamata `t.incasso()` ritorna l'incasso totale per tutti i biglietti sinora venduti per il treno `t`.

Esercizio 12

Siano `A`, `B`, `C` e `D` distinte classi polimorfe. Si considerino le seguenti definizioni.

```
template<class X>
X& fun(X& ref) { return ref; };

main() {
  B b;
  fun<A>(b);
  B* p = new D();
  C c;
  try{
    dynamic_cast<B&>(fun<A>(c));
    cout << "topolino";
  }
  catch(bad_cast) { cout << "pippo "; }
  if( !(dynamic_cast<D*>(new B())) ) cout << "pluto ";
}
```

Si supponga che:

1. il `main()` compili correttamente ed esegua senza provocare errori a run-time;
2. l'esecuzione del `main()` provochi in output su `cout` la stampa `pippo pluto`.

In tali ipotesi, per ognuna delle relazioni di sottotipo $X \leq Y$ nelle seguenti tabelle segnare con una croce l'entrata

(a) "Vero" per indicare che `X` **sicuramente è** sottotipo di `Y`;

(b) "Falso" per indicare che `X` **sicuramente non è** sottotipo di `Y`;

(c) "Possibile" **altrimenti**, ovvero se non valgono nè (a) nè (b).

	Vero	Falso	Possibile
$A \leq B$			
$A \leq C$			
$A \leq D$			
$B \leq A$			
$B \leq C$			
$B \leq D$			

	Vero	Falso	Possibile
$C \leq A$			
$C \leq B$			
$C \leq D$			
$D \leq A$			
$D \leq B$			
$D \leq C$			

Esercizio 13

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 {};
```

Assumiamo 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` ed in cui tutte le componenti che sono una entrata di menu e sono contenute in almeno 2 contenitori vengono disabilitate.

2. Se invece `c` non contiene nessuna componente `Button` allora solleva una eccezione di tipo `NoButton`.