Programmazione 2
Appello d'esame – 19/12/2003

Nome..... Cognome.....

Matricola..... Laurea in.....

Non si possono consultare appunti e libri. Dove previsto scrivere CHIARAMENTE la risposta nell'apposito spazio. ATTENZIONE: In tutti gli esercizi si intende la compilazione standard `g++` con il flag `-fno-elide-constructors`.

1. Si considerino le seguenti definizioni e dichiarazioni:

```
class Z {
public:
  int val;
  Z(int x=0): val(x) {}
};

class B {
protected:
  Z* punt;
public:
  B(Z* p = 0): punt(p) {}
};

class D: public B {
private:
  int size;
public:
  D(int x = 5): B(new Z[x]), size(x) {
   for(int i=0; i<size; ++i) (punt+i)->val=x;
  }
  class iteratore {
  // ...
  public:
    bool operator==(const iteratore&) const;
    iteratore operator++();
  };
  D(const D&);             // copia profonda
  D& operator=(const D&);  // assegnazione profonda
  ~D();                    // distruzione profonda

  iteratore begin() const; // iteratore iniziale
  iteratore end() const;   // iteratore finale
};
```

   Quindi gli oggetti della classe `D` derivata da `B` rappresentano array dinamici di oggetti di `Z` dove il campo dati `size` rappresenta la dimensione dell'array. Si ridefinisca l'assegnazione, il costruttore di copia ed il distruttore di `D` in modo da gestire la memoria **senza condivisione**: quindi, assegnazione profonda, costruttore di copia profonda e distruzione profonda.

   Inoltre, si definisca la classe `iteratore` interna a `D` i cui oggetti rappresentano degli iteratori sull'array dinamico rappresentato da un oggetto di `D` (suggerimento: fare attenzione ai campi dati necessari per rappresentare precisamente un iteratore, un semplice puntatore non basta...). La classe `D` rende disponibili i metodi `begin()` ed `end()` che ritornano gli iteratori iniziale e finale sull'oggetto di invocazione. La classe `iteratore` rende disponibili l'operatore di uguaglianza tra iteratori e l'operatore di incremento prefisso che sposta l'oggetto iteratore di invocazione alla posizione successiva se questa esiste altrimenti lo fa diventare l'iteratore finale.

2. Si consideri la seguente classe `C`.

```
class B {
protected:
  int i;
public:
  virtual ~B() {}
  B(int x = 5): i(x) {}
  int f() {return i;}
  virtual int m(int x) {return i+x;}
  int f(int x) {return i-x;}
};
```

Per ciascuna delle seguenti classi `D` derivate da `B` con corrispondente `main()` si scriva nell'apposito spazio delimitato dai puntini **NON COMPILA** per indicare la mancata compilazione (o della classe `D` oppure del `main()`) mentre in caso di corretta compilazione si scriva nell'apposito spazio la stampa prodotta in output (è un solo numero) dall'esecuzione del `main()`.

```
class D: public B {
private: int k;
public:
  D(int x = 3): B(x), k(x) {}
  int m(B b) {return k + b.i;}
};
```

```
main() {B b; D d; cout << d.m(b);}

.....................................
```

```
class D: public B {
private: int k;
public:
  D(int x = 3): B(x), k(x) {}
  int m(D d) {return k + d.i;}
};
```

```
main() {D d1, d2(4); cout << d1.m(d2);}

.....................................
```

```
class D: public B {
private: int k;
public:
  D(int x = 3): B(x), k(x) {}
  void m(int x) {cout << k + x;}
};
```

```
main() { D d; d.m(6); }

.....................................
```

```
class D: public B {
private: int k;
public:
  D(int x = 3): B(x), k(x) {}
  int f(int x) {return x + i;}
};
```

```
main() {D d(2); cout << d.f(3);}

.....................................
```

```
class D: public B {
private: int k;
public:
  D(int x = 3): B(x), k(x) {}
  double f(double x) {return k+x;}
};
```

```
main() { D d; cout << d.f(); }

.............................
```

```
class D: public B {
private: int k;
public:
  D(int x = 3): B(x), k(x) {}
  int f() {B b(7); return k + b.i;}
};
```

```
main() {D d(5); cout << d.f();}

.............................
```

```
class D: public B {
private: int k;
public:
  D(int x = 3): B(x), k(x) {}
  void f(int x) {D d(4); cout << k+d.i;}
};
```

```
main() { D d(5); d.f(2); }




.........................
```

3. Si considerino le seguenti due classi:

```
class B {
protected:
  int i;
public:
  virtual ~B() {}
  B(int x=1): i(x) {}
  int quadrato() const {return i*i;}
};
```

```
class C: public B {
public:
  C(int x=2): B(x) {}
  int cubo() const {return i*i*i;}
};
```

Si definisca una classe `Esercizio` che soddisfa le seguenti specifiche (**attenzione:** non è permessa alcuna modifica alle classi `B` e `C`).

(a) Un oggetto della classe `Esercizio` è caratterizzato da un set `s` di puntatori a `B`, cioè da un oggetto di tipo `set<B*>`.

(b) `Esercizio` rende disponibile un metodo `aggiungi(B& b)` che inserisce nel set `s` dell'oggetto di invocazione un puntatore all'oggetto `b`.

(c) `Esercizio` rende disponibile un metodo `Fun()` che ritorna l'intero k calcolato come segue:

- Sia `p` un puntatore non nullo ad un oggetto di `B` e sia `i` il campo dati intero di tale oggetto. Definiamo l'intero $f(\mathtt{p})$ come segue: se `D*` è il tipo dinamico di `p` e `D` è un sottotipo di o uguale a `C` allora $f(\mathtt{p})$ è il cubo di `i`; altrimenti, $f(\mathtt{p})$ è il quadrato di `i`.
- Sia `s` il set dell'oggetto di invocazione di `Fun()`. Allora `Fun()` ritorna

$$k = \sum_{\mathtt{p} \in \mathtt{s}} f(\mathtt{p}).$$

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

```
main() {
  B b1, b2(3); C c1, c2(3), c3(-2);
  Esercizio e;
  e.aggiungi(b1); e.aggiungi(b2); e.aggiungi(c1); e.aggiungi(c2); e.aggiungi(c3);
  e.aggiungi(c2); e.aggiungi(b2); // i puntatori a c2 e b2 sono gia' presenti
  cout << e.Fun(); // stampa: 37 = 1*1 + 3*3 + 2*2*2 + 3*3*3 + -2*-2*-2
}
```

4.
```
#include<iostream>
#include<math.h>
using namespace std;

class B {
protected:
  int i;
public:
  virtual ~B() {}
  B(int x = 5): i(x) {}
  int valore() const {return i;}
  virtual int valoreAssoluto() const { return abs(valore()); }
};

class D: public B {
private:
  int k;
public:
  D(int x = -3): k(x) {}
  D(const D& d): k(d.k) {cout << k << " Dc ";}
  int valore() const {return i+k;}
  virtual int valoreAssoluto() const { return abs(valore()); }
};

D* Fun(const B& b) { return new D(b.valoreAssoluto()); }

main() {
  B b1, b2(-4); D d1, d2(-7);
  b1 = *(Fun(d1)); cout << b1.valore() <<' '<< b1.valoreAssoluto() <<" UNO\n";
  B* p = Fun(b2); cout << p->valore() <<' '<< p->valoreAssoluto() <<" DUE\n";
  B& b3 = *(Fun(d2)); cout << b3.valore() <<' '<< b3.valoreAssoluto() <<" TRE\n";
  cout << Fun(d2)->valore() << ' ' << Fun(d2)->valoreAssoluto() <<" QUATTRO\n";
  D& d3 = *(Fun(b2)); cout << d3.valore() <<' '<< d3.valoreAssoluto() <<" CINQUE\n";
  B* q = new B(d2); cout << q->valore() <<' '<< q->valoreAssoluto() <<" SEI\n";
  D* r = new D(d2); cout << r->valore() <<' '<< r->valoreAssoluto() <<" SETTE";
}
```

Questo programma compila correttamente (si ricorda che la funzione `abs(int i)` definita nel file header standard `math.h` ritorna il valore assoluto dell'intero `i`). Quali stampe produce la sua esecuzione? Se una riga non produce alcuna stampa (oltre a quella già indicata) si scriva **NESSUNA STAMPA**.

..... `UNO`

..... `DUE`

..... `TRE`

..... `QUATTRO`

..... `CINQUE`

..... `SEI`

..... `SETTE`

5. Il seguente programma compila correttamente. Si scrivano negli appositi spazi le stampe prodotte dalla sua esecuzione. Se una riga non produce alcuna stampa (oltre a quella già indicata) si scriva **NESSUNA STAMPA**.

```
class Z {
public:
  Z(int x=1) {cout << x << " Z01 ";}
  Z(const Z& z) {cout << "Zc ";}
  Z& operator=(const Z& z) {cout << "Z= ";}
};

class B {
protected:
  Z z;
public:
  B(int x = 5): z(x) {cout << x << " B01 ";}
  B(const B& b): z(6) {cout << "Bc ";}
  B& operator=(const B& b) {
    z=Z(b.z); cout << "B= ";
  }
};

class D: public B {
private:
  Z* p;
public:
  D(): B(4) {p=new Z(3); cout << "D0 ";}
  D(int x): p(new Z[x]) {cout << "D1 ";}
  D(const D& d): p(d.p) {cout << "Dc ";}
  D& operator=(const D& d) {
   z=d.z; p=d.p; cout << "D= ";
  }
};

D Fun(D d, B& b) { b=d; return d; }

main() {
 B b1; cout<<"**1\n"; ..........................................................**1

 D d1; cout<<"**2\n"; ..........................................................**2

 D d2(2); cout<<"**3\n"; .......................................................**3

 d2 = Fun(d1,b1); cout<<"**4\n"; ...............................................**4

 B b2 = d2; cout<<"**5\n"; .....................................................**5

 B b3 = Fun(d1,d1); cout<<"**6"; ...............................................**6
}
```