Programmazione ad Oggetti
Appello d'Esame – 9/12/2008

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

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

Non si possono consultare appunti e libri. Dove previsto scrivere CHIARAMENTE la risposta nell'apposito spazio.

Esercizio 1

```
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 nell'apposito spazio le stampe provocate in output dall'esecuzione del `main()` (nel caso in cui una certa riga non contenga alcuna stampa si scriva **NESSUNA STAMPA**).

..... `**1`

..... `**2`

..... `**3`

..... `**4`

Esercizio 2

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.

Esercizio 3

```
class D;

class B {
public:
  virtual D* f() =0;
};

class C {
public:
  virtual C* g();
  virtual B* h() =0;
};

class D: public B, public C {
public:
  D* f() {cout << "D::f ";  return new D;}
  D* h() {cout << "D::h "; return dynamic_cast<D*>(g());}
};

C* C::g() {
  cout << "C::g ";
  B* p = dynamic_cast<B*>(this);
  if(p) return p->f(); else return this;
}

class E: public D {
public:
  E* f() {
    cout << "E::f ";
    E* p = dynamic_cast<E*>(g());
    if(p) return p; else return this;
  }
};

class F: public E {
public:
  E* g() {cout << "F::g "; return new F;}
  E* h() {
    cout << "F::h ";
    E* p = dynamic_cast<E*>(E::g());
    if(p) return p; else return new F;
  }
};

B* p; C* q; D* r;
```

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. Si scriva nell'apposito spazio contiguo:

- **NON COMPILA** quando tale `main()` non compila;
- **ERRORE RUN-TIME** quando tale `main()` compila ma l'esecuzione provoca un errore run-time;
- la stampa che produce in output su `cout` nel caso in cui tale `main()` compili ed esegua senza errori; se non provoca alcuna stampa si scriva **NESSUNA STAMPA**.

`p = new E; p->h();`	
`p = new E; p->f();`	
`p = new D; (dynamic_cast<D*>(p))->h();`	
`q = new D; q->g();`	
`q = new E; q->h();`	
`q = new F; q->g();`	
`r = new E; r->f();`	
`r = new F; r->f();`	
`r = new F; r->g();`	
`r = new F; r->h();`	

Esercizio 4

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`.