Esercizio 2:

9 A01 ZERO
5 A01 UNO
3 A01 DUE
TRE
5 A01 5 A01 7 A01 B0 QUATTRO
5 A01 5 A01 Bc Ac Ac CINQUE
5 A01 5 A01 Bc Ac Ac SEI

Esercizi di Programmazione ad Oggetti

Lista n. 1

Esercizio 1

Definire una classe `IntMod` i cui oggetti rappresentano numeri interi modulo un dato intero n, che deve essere dichiarato come campo dati statico.
Definire metodi statici di `set_modulo()` e `get_modulo()` per tale campo dati statico.
Devono essere disponibili gli operatori di somma e moltiplicazione tra oggetti di `IntMod`.
Definire inoltre opportuni convertitori di tipo affiché questa classe sia liberamente usabile assieme al tipo primitivo `int` e valga la seguente condizione:
quando in una espressione compaiono interi e oggetti di `IntMod` il tipo dell'espressione dovrà essere intero.

Scrivere infine un programma d'esempio che utilizza tutti i metodi della classe.

Esercizio 2

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

```
#include<iostream>
using std::cout;

class A {
private:
  int x;
public:
  A(int k = 5): x(k) {cout << k <<  " A01 ";}
  A(const A& a): x(a.x) {cout << "Ac ";}
  A g() const {return *this;}
};

class B {
private:
  A ar[2];
  static A a;
public:
  B() {ar[1] = A(7); cout << "B0 ";}
  B(const B& b) {cout << "Bc ";}
};
A B::a = A(9);

A Fun(A* p, const A& a, B b) {
  *p = a;
  a.g();
  return *p;
};

main() {
  cout << ``ZERO\n'';
  A a1; cout << "UNO\n";
  A a2(3); cout << "DUE\n";
  A* p = &a1; cout << "TRE\n";
  B b; cout << "QUATTRO\n";

  a1 = Fun(p,a2,b); cout << "CINQUE\n";

  A a3 = Fun(&a1,*p,b); cout << "SEI";
}
```

Esercizio 3

Perché il seguente programma non compila? Modificare o eliminare una e soltanto una delle righe 1-8 in modo che il programmi compili.

```
class C {
public:                          // 1
  int *const p;                  // 2
  C(int a=0): p(new int(a))      // 3
  { }                            // 4
};

main() {
  C x(3);                        // 5
  C y;                           // 6
  x=y;                           // 7
  C z(y);                        // 8
}
```

Esercizio 4

0 6 7 8

Il seguente programma compila ed esegue correttamente. Quale stampa di output provoca?

```
#include<iostream>
#include<string>
using std::string; using std::cout;

class C {
private:
  int d;
public:
  C(string s=""): d(s.size()) {}
  explicit C(int n): d(n) {}
  operator int() {return d;}
  C operator+(C x) {return C(d+x.d);}
};

main() {
  C a, b("pippo"), c(3);
  cout << a << ' ' << 1+b << ' ' << c+4 << ' ' << c+b;
}
```

Esercizio 5

Definire, separando interfaccia ed implementazione, una classe `Raz` i cui oggetti rappresentano un numero razionale $\frac{num}{den}$ (naturalmente, i numeri razionali hanno sempre un denominatore diverso da 0). La classe deve includere:

1. opportuni costruttori;
2. un metodo `Raz inverso()` con il seguente comportamento: se l'oggetto di invocazione rappresenta $\frac{n}{m}$ allora `inverso` ritorna un oggetto che rappresenta $\frac{m}{n}$;
3. un operatore esplicito di conversione al tipo primitivo `double`;
4. l'overloading come metodi interni degli operatori di somma e moltiplicazione;

5. l'overloading come metodo interno dell'operatore di incremento postfisso, che, naturalmente, dovrà incrementare di 1 il razionale di invocazione;
6. l'overloading dell'operatore di output su `ostream`;
7. un metodo statico `Raz uno()` che ritorna il razionale 1.

Definire un esempio di `main()` che usi tutti i metodi della classe.

B012 UNO
Bc B012 Bc b C012 DUE
B012 B012 Bc b C012 TRE
Bc Bc QUATTRO
ebd CINQUE

Esercizio 6

Il seguente programma compila ed esegue correttamente. Quali stampe provoca in output?

```
#include<iostream>
#include<string>
using std::string; using std::cout;

class B {
public:
  string s;
  B(char x='a', char y='b') {s += x; s += y; cout << "B012 ";}
  B(const B& obj): s(obj.s) {cout << "Bc "; }
};

class C {
private:
  B t;
  B* p;
  B u;
public:
  string s;
  C(char x='c', B y = B('d')): u(y), s(y.s) {
    s += x;
    cout << s[s.size()-2] << " C012 ";
  }
};

B F(B x, C& y) {
  (x.s) += (y.s)[0];
  return x;
}

main() {
  B b('e'); cout << "UNO\n";
  C c1('f',b); cout << "DUE\n";
  C c2; cout << "TRE\n";
  b=F(b,c2); cout <<"QUATTRO\n";
  cout << b.s  << " CINQUE";
}
```

2 5 2 0x7ffee4ff1194 0x7ffee4ff1194

Esercizio 7

Si consideri il seguente programma.

```
#include<iostream>
using std::cout;

class C {
```

```
public:
  int x;
  C(int k=5): x(k) {};
  C* m(C& c) {
    if((c.x != 5) && (x==5)) return &c;
    return this;
  }
};

main() {
  C a, b(2), c(a);
  cout << (b.m(b))->x << ' ' << (a.m(a))->x << ' ' << (b.m(c))->x
       << ' ' << c.m(a) << ' ' << c.m(c);
}
```

Il seguente programma compila correttamente? Se sì, al sua esecuzione quali stampe provoca in output?

Esercizio 8

Definire, separando interfaccia ed implementazione, una classe `Data` i cui oggetti rappresentano una data con giorno della settimana (lun-mar-...-dom). La classe deve includere:

- opportuni costruttori
- metodi di selezione per ottenere giorno della settimana, giorno, mese, anno di una data
- l'overloading dell'operatore di output esternamente alla classe
- l'overloading dell'operatore di uguaglianza
- l'overloading dell'operatore relazionale < che ignori il giorno della settimana
- un metodo `aggiungi_uno()` che avanza di un giorno la data di invocazione. Esempi: `lun 21/10/2002 => mar 22/10/2002`; `gio 31/1/2002 => ven 1/2/2002`; `mar 31/12/2002 => mer 1/1/2003`. Ignorare gli anni bisestili

Esemplificare l'uso della classe e di tutti i suoi metodi tramite un esempio di `main()`.

Esercizi di Programmazione ad Oggetti

UNO
N2 N2 N2 N2 DUE
pluto~N paperino~N topolino~N pippo~N ~C TRE

Codice corretto: https://pastebin.com/mtWL4Da3

Lista n. 2

Esercizio 1

Si consideri il seguente programma.

```
class S {
public:
  string s;
  S(string t): s(t) {}
};
class N {
private:
  S x;
public:
  N* next;
  N(S t, N* p): x(t), next(p) {cout << "N2 ";}
  ~N() {if (next) delete next; cout << x.s + "~N ";}
};
class C {
  N* pointer;
public:
  C(): pointer(0) {}
  ~C() {delete pointer; cout << "~C ";}
  void F(string t1, string t2 = "pippo") {
    pointer = new N(S(t1),pointer); pointer = new N(t2,pointer);
  }
};
main(){
  C* p = new C; cout << "UNO\n";
  p->F("pluto","paperino"); p->F("topolino"); cout <<"DUE\n";
  delete p; cout <<"TRE\n";
}
```

Il programma compila correttamente? Se sì, quali stampe provoca in output?

Esercizio 2

0 1 2 3 UNO
0 1 3 6 DUE

Si consideri il seguente programma.

```
#include<iostream>
using std::cout;

class It {
  friend class C;
public:
  bool operator<(It i) {return index < i.index;}
  It operator++(int) { It t = *this; index++; return t; }
  It operator+(int k) {index = index + k; return *this; }
private:
  int index;
};
class C {
public:
  C(int k) {
    if (k>0) {dim=k; p = new int[k];}
```

```
    for(int i=0; i<k; i++) *(p+i)=i;
  }
  It begin() { It t; t.index = 0; return t; }
  It end() { It t; t.index = dim; return t; }
  int& operator[](It i) {return *(p + i.index);}
private:
  int* p;
  int dim;
};

main() {
  C c1(4), c2(8);
  for(It i = c1.begin(); i < c1.end(); i++) cout << c1[i] << ' ';
  cout << "UNO\n";
  It i = c2.begin();
  for(int n=0; i < c2.end(); ++n, i = i+n) cout << c2[i] << ' ';
  cout << "DUE\n";
}
```

Il programma compila correttamente? Se sì, quali stampe provoca in output?

Esercizio 3

https://pastebin.com/CWZemjYE

Si considerino le seguenti dichiarazioni e definizioni:

```
class Nodo {
private:
  Nodo(string st="***", Nodo* s=0, Nodo* d=0): info(st), sx(s), dx(d) {}
  string info;
  Nodo* sx;
  Nodo* dx;
};
class Tree {
public:
  Tree(): radice(0) {}
  Tree(const Tree&); // dichiarazione costruttore di copia
private:
  Nodo* radice;
};
```

Quindi, gli oggetti della classe `Tree` rappresentano *alberi binari ricorsivamente definiti di stringhe*. Si ridefinisca il costruttore di copia di `Tree` in modo che esegua copie profonde. Scrivere esplicitamente eventuali dichiarazioni `friend` che dovessero essere richieste da tale definizione.

Esercizio 4

Definire una classe `Vettore` i cui oggetti rappresentano array di interi. `Vettore` deve includere un costruttore di default, l'overloading dell'uguaglianza, dell'operatore di output e dell'operatore di indicizzazione. Inoltre deve includere un costruttore di copia "profonda" e l'overloading dell'assegnazione come assegnazione "profonda".

Esercizio 5

```
class C {
public:
    C(): size(1), a(new int[1]) {a[0]=0;}
    C& operator=(const C& x) {
```

1 2 UNO
1 2 DUE
1 5 TRE
1 5 QUATTRO
4 2 CINQUE
7 5 SEI
4 6 SETTE
4 2 OTTO
4 2 ~C NOVE
4 6 ~C 7 5 ~C

```
            if(this!=&x){
              size=x.size;
              a=new int[size];
              for(int i=0;i<size;i++) a[i]=x.a[i];
            }
            return *this;
    }
    void add(int k) {
      int *b=a;
            a=new int[size+1];
            ++size;
            a[0]=k;
            for(int i=1;i<size;i++) a[i]=b[i-1];
            delete[] b;
    }
    int& operator[](int i) const {return a[i];}
    void stampa() const {
            for(int i=0;i<size;i++) cout<<a[i]<< ' ';
    }
    ~C() {stampa(); cout<<"~C "; delete[] a;}
private:
    int size;
    int* a;
};

main(){
    C v; v.add(1);
    C w=v; w[1]=2;
    v.stampa(); cout<<"UNO\n";
    w.stampa(); cout<<"DUE\n";
    C* p=new C; p->add(3);
    *p=v;
    (*p)[0]=4; v[1]=5;
    v.stampa(); cout<<"TRE\n";
    w.stampa(); cout<<"QUATTRO\n";
    p->stampa(); cout<<"CINQUE\n";
    w=*p;
    w[1]=6; v[0]=7;
    v.stampa(); cout<<"SEI\n";
    w.stampa(); cout<<"SETTE\n";
    p->stampa(); cout<<"OTTO\n";
    delete p; cout<<"NOVE\n";
}
```

Il precedente programma compila correttamente. Quali stampe provoca la sua esecuzione?

Esercizi di Programmazione ad Oggetti

Lista n. 3

Esercizio 1

Definire un template di classe contenitore `Dizionario<T>` i cui oggetti rappresentano una collezione di coppie (chiave,valore) dove la chiave è una stringa ed il valore è di tipo `T`. Ad una certa stringa può essere associato un solo valore di `T`. Si tratta quindi di definire un template di classe analogo al contenitore STL `map<string,T>`. Dovranno essere disponibili le seguenti funzionalità:

- inserimento: `bool insert(string, const T&)`
- rimozione: `bool erase(string)`
- ricerca per chiave: `T* findValue(string)`
- ricerca per valore: `vector<string> findKey(const T&)`
- overloading degli operatori di indicizzazione e output

Esercizio 2

```
class Z {
public:
  ...
};

template <class T1, class T2=Z>
class C {
public:
  T1 x;
  T2* p;
};

template<class T1,class T2>
void fun(C<T1,T2>* q) {
  ++(q->p);
  if(true == false) cout << ++(q->x);
  else cout << q->p;
  (q->x)++;
  if(*(q->p) == q->x) *(q->p) = q->x;
  T1* ptr = &(q->x);
  T2 t2 = q->x;
}

main(){
  C<Z> c1; fun(&c1); C<int> c2; fun(&c2);
}
```

Si considerino le precedenti definizioni. Fornire una dichiarazione (non è richiesta la definizione) dei membri pubblici della classe `Z` nel **minor numero possibile** in modo tale che la compilazione del precedente `main()` non produca errori. **Attenzione:** ogni dichiarazione in `Z` non necessaria per la corretta compilazione del `main()` sarà penalizzata.

```
class Z {
public:
  int operator++() {} // 1
  Z& operator++(int) {} // 2
  bool operator==(const Z& x) const {} // 3
  Z(int x=0) {} // 4
```

```
};

template<class T1,class T2>
void fun(C<T1,T2>* q) {
  ++(q->p); // nulla
  if(true == false) cout << ++(q->x); // 1
  else cout << q->p;  // nulla
  (q->x)++; // 2
  if(*(q->p) == q->x) // 3
    *(q->p) = q->x; // nulla
  T1* ptr = &(q->x); // nulla
  T2 t2 = q->x; // 4 conversione T1 => T2
}
```

Esercizi di Programmazione ad Oggetti

Lista n. 4

Esercizio 1

Definire una superclasse `ContoBancario` e due sue sottoclassi `ContoCorrente` e `ContoDiRisparmio` che soddisfano le seguenti specifiche:

- Ogni `ContoBancario` è caratterizzato da un saldo e rende disponibili due funzionalità di deposito e prelievo: `int deposita(int)` e `int preleva(int)` che ritornano il saldo aggiornato dopo l'operazione di deposito/prelievo.
- Ogni `ContoCorrente` è caratterizzato anche da una spesa fissa uguale per ogni `ContoCorrent` che deve essere detratta dal saldo ad ogni operazione di deposito e prelievo.
- Ogni `ContoDiRisparmio` deve avere un saldo non negativo e pertanto non tutti i prelievi sono permessi; d'altra parte, le operazioni di deposito e prelievo non comportano costi aggiuntivi. e restituiscono il saldo aggiornato.
- Si definisca inoltre una classe `ContoArancio` derivata da `ContoDiRisparmio`. La classe `ContoArancio` deve avere un `ContoCorrente` di appoggio: quando si deposita una somma S su un `ContoArancio`, S viene prelevata dal `ContoCorrente` di appoggio; d'altra parte, i prelievi di una somma S da un `ContoArancio` vengono depositati nel `ContoCorrente` di appoggio.

Esercizi di Programmazione ad Oggetti

Lista n. 5

Esercizio 1

Definire una superclasse `Auto` i cui oggetti rappresentano generiche automobili e due sue sottoclassi `Benzina` e `Diesel`, i cui oggetti rappresentano automobili alimentate, rispettivamente, a benzina e a diesel (ovviamente non esistono automobili non alimentate e si assume che ogni auto è o benzina o diesel). Ci interesserà l'aspetto fiscale delle automobili, cioè il calcolo del bollo auto. Queste classi devono soddisfare le seguenti specifiche:

- Ogni automobile è caratterizzata da un numero di cavalli fiscali. La tassa per cavallo fiscale è unica per tutte le automobili (sia benzina che diesel) ed è fissata in 5 euro. La classe `Auto` fornisce un metodo `tassa()` che ritorna la tassa di bollo fiscale per l'automobile di invocazione.
- La classe `Diesel` è dotata (almeno) di un costruttore ad un parametro intero `x` che permette di creare un'auto diesel di `x` cavalli fiscali. Il calcolo del bollo fiscale per un'auto diesel viene fatto nel seguente modo: si moltiplica il numero di cavalli fiscali per la tassa per cavallo fiscale e si somma una addizionale fiscale unica per ogni automobile diesel fissata in 100 euro.
- Un'auto benzina può soddisfare o meno la normativa europea Euro4. La classe `Benzina` è dotata di (almeno) un costruttore ad un parametro intero `x` e ad un parametro booleano `b` che permette di creare un'auto benzina di `x` cavalli fiscali che soddisfa Euro4 se `b` vale `true` altrimenti che non soddisfa Euro4. Il calcolo del bollo fiscale per un'auto benzina viene fatto nel seguente modo: si moltiplica il numero di cavalli fiscali per la tassa per cavallo fiscale; se l'auto soddisfa Euro4 allora si detrae un bonus fiscale unico per ogni automobile benzina fissato in 50 euro, altrimenti non vi è alcuna detrazione.

Si definisca inoltre una classe `ACI` i cui oggetti rappresentano delle filiali ACI addette all'incasso dei bolli auto. Ogni oggetto ACI è caratterizzato da un vector di puntatori ad auto, cioè un oggetto `vector<Auto*>`, che rappresenta la lista delle automobili gestite da una filiale ACI. La classe `ACI` fornisce un metodo `aggiungiAuto(const Auto& a)` che aggiunge l'auto `a` alla lista gestita dalla filiale di invocazione. Inoltre, la classe `ACI` fornisce un metodo `incassaBolli()` che ritorna la somma totale dei bolli che devono pagare tutte le auto gestite dalla filiale di invocazione.

Definire infine un esempio di `main()` in cui viene costruita una filiale ACI a cui vengono aggiunte quattro automobili in modo tale che l'incasso dei bolli ammonti a 1600 euro.

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.

Esercizi di Programmazione ad Oggetti

Lista n. 7

Esercizio 1

Si consideri la seguente gerarchia di classi.

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

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

class C: virtual public A, virtual public B {
public:
  C() {cout << "C0 ";}
  C(B* x, A y = A()) {cout << "C1-2 ";}
};

class D: public C {
public:
  D() {cout << "D0 ";}
  D(C x) {cout << "D1 ";}
};
```

Si considerino le seguenti definizioni di variabili globali ed i seguenti `main()` composti di una singola istruzione. Per ognuno dei `main()` determinare se compila o meno e la stampa che produce in output nel caso invece compili.

```
A a; B b; C c; D d;

main(){D z0(a);}

main(){D z1(b);}

main(){D z2(&b);}

main(){D z3(&c);}

main(){D z4(c);}

main(){D z5(d);}

main(){C z6(&a,c);}

main(){C z7(&c);}

main(){C z8(&d,d);}
```

Esercizio 2

Si consideri la seguente gerarchia di classi e le seguenti variabili globali:

```
class A {public: virtual ~A() {} };
class B: virtual public A {};
class C: virtual public A {};
class D: virtual public A {};
class E: virtual public C {};
class F: virtual public C {};
class G: public B, public E, public F {};

B b; D d; E e; F f; G g; A* pa; B* pb; C* pc; F* pf;
```

Per ognuno dei `main()` determinare se compila o meno e la stampa che produce in output nel caso invece compili.

```
main() {pc = &e; cout << (dynamic_cast<D*> (pc) ? "OK" : "NO");}

main() {cout << (dynamic_cast<B*> (&g) ? "OK" : "NO");}
```

```
main() {pa = &f; cout << (dynamic_cast<C*> (pa) ? "OK" : "NO");}

main() {pb = &b; cout << (dynamic_cast<G*> (pb) ? "OK" : "NO");}

main() {cout << (dynamic_cast<D*> (&d) ? "OK" : "NO");}

main() {pf = &g; cout << (dynamic_cast<E*> (pf) ? "OK" : "NO");}

main() {pf = &f; cout << (dynamic_cast<E*> (pf) ? "OK" : "NO");}
```

Esercizio 3

Si considerino le seguenti definizioni di classe e funzione:

```
class A {
public:
  virtual ~A() {};
};
class B: public A {};
class C: virtual public B {};
class D: virtual public B {};
class E: public C, public D {};

char F(A* p, C& r) {
  B* punt1 = dynamic_cast<B*> (p);
  try{
    E& s = dynamic_cast<E&> (r);
  }
  catch(bad_cast) {
    if(punt1) return 'O';
    else return 'M';
  }
  if(punt1) return 'R';
  return 'A';
}
```

Si consideri inoltre il seguente `main()` incompleto, dove `?` è semplicemente un simbolo per una incognita:

```
main(){
  A a; B b; C c; D d; E e;
  cout << F(?,?) << F(?,?) << F(?,?) << F(?,?);
}
```

Definire opportunamente le chiamate in tale `main()` usando gli oggetti `a`, `b`, `c`, `d`, `e` locali al `main()` in modo tale che la sua esecuzione provochi la stampa `ROMA`.

Esercizio 4

```
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();`	

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`.