Facoltà di Ingegneria
Università di Udine
Dispense del corso di
INGEGNERIA DEL SOFTWARE
A. A. 2009-10

Compiti d'esame svolti

a cura di
Andrea Schaerf

25 maggio 2010

Introduzione

Programma del corso

Introduzione all'ingegneria del software

- Ciclo di vita del software
- Qualità dei programmi

Introduzione al C++

- Richiami di C e introduzione al C++
- Funzioni in C++
- Puntatori e array dinamici
- Utilizzo del compilatore GNU C++ in ambiente `cygwin` e sue funzionalità di base

Le classi e gli oggetti in C++

- Classi e oggetti
- Costruttori e distruttori
- *Overloading* di operatori
- Funzioni speciali
- Composizione di classi
- I *template*
- La STL (*Standard Template Library*)
  - Caratteristiche generali
  - Le stringhe (la classe `string`)
  - I vettori (il *template* `vector`)

Analisi del software

- UML (Unified Modeling Language) come strumento di analisi
- Diagramma delle classi in UML
- Diagramma degli stati in UML
- Cenni ad altri diagrammi in UML

Progettazione del software

- Dalla specifica UML di una classe alla sua realizzazione in C++
- Realizzazioni di associazioni UML in C++
- Realizzazioni della relazione *part-of*
- Utilizzo della STL nelle realizzazioni
- Progetto di applicazioni complesse

Modalità d'esame

L'esame di Ingegneria del Software si compone di 3 parti: compito scritto, prova a casa e discussione orale:

Compito scritto: Il compito scritto consiste in un esercizio di analisi e progettazione utilizzando UML e C++. Durante il compito, la cui durata è di 2-3 ore, non è consentito consultare libri, appunti, dispense o qualsiasi altro materiale e non è consentito uscire dall'aula.

La valutazione dello scritto verrà resa nota solo *dopo* la discussione dello scritto e dell'elaborato svolto a casa, quindi tutti gli studenti che hanno sostenuto lo scritto sono implicitamente ammessi a sostenere l'orale.

Prova a casa: Nel tempo che intercorre tra il compito scritto e la prova orale (tipicamente 3-4 giorni) lo studente è tenuto a compilare ed eseguire la soluzione del compito.

La soluzione proposta a casa deve essere tendenzialmente la stessa di quella svolta nel compito, e la sua correttezza deve essere verificata scrivendo i necessari *driver* e *stub* C++ e usando dei dati di test scelti opportunamente. Nel caso la soluzione svolta in classe non sia corretta, questa deve essere modificata opportunamente fino ad ottenere il corretto funzionamento su tutti i dati di test.

Nel caso lo studente non abbia svolto parte dell'esercizio nel compito, dovrà comunque portare la soluzione svolta a casa; tale soluzione può ovviamente essere qualsiasi.

Scopo principale della prova a casa è quello di fare una autovalutazione precedente all'orale, che porta ad una discussione (e valutazione) dello scritto più proficua. In particolare, tale soluzione serve allo studente per dimostrare che, a parte eventuali modifiche, la soluzione proposta nel compito è corretta.

Discussione orale: Lo studente deve presentarsi all'orale obbligatoriamente con una memoria contenente i file che compongono la nuova soluzione e con la stampa dei file stessi e dei test effettuati.

La discussione orale è costituita essenzialmente da una discussione del compito scritto e della soluzione svolta a casa dallo studente (più eventuali domande aggiuntive a discrezione del docente).

Materiale didattico

Libri di testo consigliato

- Cay Horstmann. Fondamenti di C++. McGraw-Hill, 2003.
- Martin Fowler. UML distilled, guida rapida al linguaggio di modellazione standard (3ed), Pearson, 2004.

Materiale disponibile via rete e su CD

All'indirizzo `http://www.diegm.uniud.it/schaerf/didattica.php` sono disponibili:

Compiti d'esame: La versione elettronica di tutti i programmi contenuti in questa dispensa e di quelli che si sono aggiunti successivamente.

Esercitazioni: Il testo e le soluzioni di tutte le esercitazioni sia di *azzeramento* che proprie del corso.

Altro materiale: Registro delle lezioni con i programmi svolti in aula, esercizi aggiuntivi ed informazioni generali sul corso.

Utilizzo del compilatore `gcc` in ambiente `cygwin` sotto Windows

Installazione di `cygwin`

- Reperire i file contenenti il materiale del corso dal docente
- Eseguire il file `setup.exe` nell'apposita cartella. Si proceda selezionando *Install from local directory* nella prima finestra di dialogo, e accettando tutti i default proposti dal programma di installazione. Quando appare la finestra in cui appaiono i pacchetti da installare, selezionare dalla categoria *Devel* i pacchetti `gcc`, `gdb` e `make`. Per tutti gli altri pacchetti accettare la scelta di default. L'installazione richiede pochi minuti.
- Verifica dell'installazione: Al termine dell'installazione il programma avrà creato un'icona (presente sia sul desktop che sul menù Avvio) con nome Cygwin. Cliccando su tale icona si apre una finestra con interfaccia a linea di comandi il cui *prompt* è il carattere `'$'`. A questo punto, per verificare se il compilatore è stato installato correttamente si può digitare `g++ -v` e si otterrà un messaggio del tipo:

```
Reading specs from ...
gcc version 3....
```

Installazione di `Notepad++`

Eseguire il file `npp.5.6.7.Installer.exe`. Si proceda selezionando la cartella di installazione nella prima finestra di dialogo, e accettando tutti i default proposti dal programma di installazione. Anche questa installazione richiede pochi minuti.

Nota: L'uso dell'editor di testi `Notepad++` non è obbligatorio, ma semplicemente consigliato. Qualsiasi altro editor che permetta di salvare i programmi in formato testo va altrettanto bene.

Commenti generali agli esercizi e alle soluzioni

Il programma del corso è soggetto a modifiche da un anno accademico al successivo, quindi i compiti d'esame degli A.A. passati non coprono necessariamente tutte le possibili tipologie dei compiti che verranno presentati agli studenti nell'A.A. 2008-09. Di contro, alcuni esercizi di questa dispensa potranno invece risultare difficilmente risolvibili. Sarà cura del docente segnalare queste differenze in aula e durante il ricevimento studenti.

Per migliorare l'efficacia didattica alcuni dei testi d'esame sono stati modificati rispetto a quelli proposti in aula.

Si può facilmente constatare che le soluzioni degli esercizi d'esame possono anche avere lunghezza (e difficoltà) abbastanza diverse tra loro. Chiaramente il giudizio del docente terrà conto di tale differenza, attraverso una valutazione degli errori anche in base alla complessità del compito.

Le soluzioni proposte sono tutte corredate di un *driver* (la funzione `main()`), che permette di verificare la correttezza degli esercizi svolti. Alcuni dei *driver* hanno una struttura a menù che permette di eseguire le operazioni in qualsiasi ordine, mentre altri hanno una struttura fissa che permette di verificare il programma soltanto in alcune condizioni di funzionamento predefinite.

I *driver* a menù sono più flessibili, anche se più lunghi da scrivere, in quanto permettono di eseguire un insieme più grande di test. Di contro, i *driver* a struttura fissa hanno il vantaggio di non richiedere all'utente di dover identificare ogni volta i test da effettuare.

Gli studenti possono scegliere autonomamente che tipo di *driver* scrivere per la loro prova a casa.

Riguardo alla divisione dei programmi in file, si può notare come nelle soluzioni le funzioni esterne sia state sempre inserite nello stesso file della funzione `main()`. Questa scelta è giustificata da ragioni di semplicità (evitare di avere troppi file), e non dalla presunzione che questi appartengano concettualmente allo stesso modulo.

Testi

Compito del 3 aprile 2001 (soluzione a pagina 44)

Si vuole progettare di una classe per realizzare una *Scacchiera*. Ogni oggetto della classe rappresenta una scacchiera quadrata su cui si possono piazzare dei *pezzi*. Le operazioni che occorre associare alla classe sono:

- Creazione di un oggetto della classe, con la specifica della dimensione (numero di righe e di colonne) della scacchiera; appena creata, tutte le caselle della scacchiera sono libere.
- Piazzamento di un pezzo p in una caselle c in posizione $\langle riga, colonna \rangle$. Se la casella indicata contiene giÃ un pezzo, quest'ultimo viene rimpiazzato dal nuovo. Se la posizione indicata per la casella c non è nella scacchiera s, cioè se *riga* e *colonna* non sono entrambe comprese tra 1 e la dimensione della scacchiera, allora l'operazione non deve eseguire alcuna azione.
- Spostamento di un pezzo in una casella adiacente: lo spostamento viene rappresentato con la posizione della casella in cui si trova il pezzo da spostare, e con la direzione della mossa. Riguardo alla direzione, si usa 1 per indicare la direzione nord, 2 per indicare est, 3 per indicare sud, 4 per indicare ovest. Ad esempio, per spostare il pezzo in posizione $(3, 1)$, cioè riga 3 e colonna 1, nella nuova posizione $(3, 2)$ si deve indicare la posizione della casella di partenza $(3, 1)$ e la direzione 2 (est). L'operazione di spostamento deve seguire queste regole:
  - se la casella di partenza è libera, e non c'è quindi alcun pezzo da spostare, l'operazione non deve effettuare alcuna azione sulla scacchiera e deve restituire 0;
  - se nella casella di partenza c'è un pezzo, ma la casella di arrivo non è libera, l'operazione non deve effettuare alcuna azione sulla scacchiera e deve restituire -1;
  - se nella casella di partenza c'è un pezzo, ma la casella di arrivo non esiste, l'operazione non deve effettuare alcuna azione sulla scacchiera e deve restituire -2;
  - in tutti gli altri casi l'operazione deve compiere la mossa sulla scacchiera, e deve restituire 1.

Esercizio 1 (punti 10) Si scriva la definizione della classe C++ `Scacchiera`, considerando di riferirsi ad una classe `Pezzo` che si suppone già realizzata, della quale però non è nota la definizione. L'unica cosa che si sa della classe `Pezzo` è che per essa è definito l'operatore di uguaglianza `==`.

La classe `Scacchiera` deve ovviamente avere anche le opportune funzioni pubbliche per ispezionarne lo stato (dimensione e posizione dei pezzi).

Esercizio 2 (punti 10) Si scrivano le definizioni delle funzioni della classe `Scacchiera`.

Esercizio 3 (punti 10) Si scriva una funzione esterna (non *friend*) alla classe `Scacchiera` che modifichi la scacchiera spostando tutti i pezzi presenti il più possibile verso il lato nord. Ad esempio, la scacchiera qui sotto a sinistra si deve modificare nella scacchiera a destra.

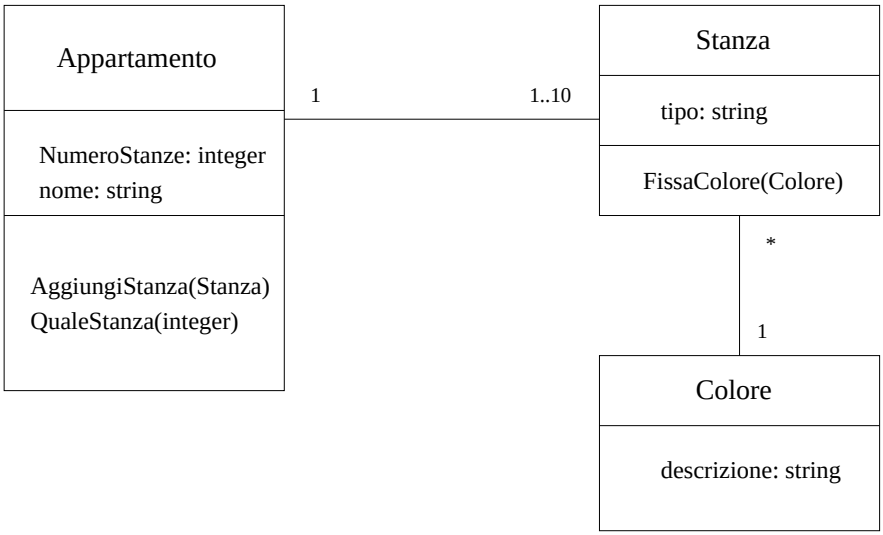
	1	2	3	4	5
1				C	
2	A				
3	B				
4		A			
5		A	A	A	

$N \uparrow$

	1	2	3	4	5
1	A	A	A	C	
2	B	A		A	
3					
4					
5					

Compito del 5 luglio 2001 (soluzione a pagina 47)

Si consideri il seguente schema concettuale in UML che riguarda la gestione di appartamenti.



La operazioni della classe *Appartamento* sono specificate in dettaglio come segue:

AggiungiStanza(s : *Stanza*) Aggiunge la stanza s all'appartamento. La stanza viene aggiunta come $n+1$-esima dell'appartamento, dove n è il numero di stanze prima dell'operazione.

QualeStanza(i : *intero*) : *Stanza* Restituisce la stanza i-esima dell'appartamento, con la precondizione che i sia minore o uguale al numero di stanze.

Esercizio 1 (punti 12) Si scrivano le definizioni delle classi C++ `Colore`, `Stanza` e `Appartamento`.

Esercizio 2 (punti 12) Si scrivano le definizioni delle funzioni delle classi. Si assuma che un appartamento al momento della creazione abbia zero stanze.

Esercizio 3 (punti 6) Si scriva una funzione esterna (non *friend* di alcuna classe) che, dato un appartamento ed un colore, fissi il colore di tutte le stanze dell'appartamento al colore dato.

Compito del 23 luglio 2001 (soluzione a pagina 50)

Si vuole progettare una classe per la gestione di ristoranti. Un ristorante ha un nome e un certo numero di tavoli, e ciascun tavolo ha un numero di posti massimo. In un dato istante, in ciascun tavolo sono presenti della persone (in numero inferiore o uguale al numero di posti del tavolo). Inoltre sono disponibili le seguenti operazioni:

AggiungiPersona(p : *Persona*, i : *intero*)**:** Aggiunge la persona p al tavolo i-esimo. Se però la persona è già presente nel ristorante, allora la funzione non esegue alcuna modifica.

LiberaTavolo(i : *intero*)**:** Tutte le persone presenti al tavolo i-esimo escono dal ristorante.

ModificaTavolo(i : *intero*, d : *intero*)**:** Il tavolo i-esimo diventa di dimensione d.
Precondizioni: Il tavolo i-esimo è vuoto.

Si assuma già disponibile la classe `Persona` di cui però non si conosce la definizione.

Esercizio 1 (punti 10) Si scriva la definizione della classe C++ `Ristorante` con le funzioni e i costruttori che si ritengono opportuni.

Esercizio 2 (punti 10) Si scrivano le definizioni delle funzioni delle classi.

Esercizio 3 (punti 10) Si consideri già realizzata la seguente classe `Gruppo`.

```
class Gruppo
{
 public:
  int QuantePersone(); // restituisce il numero di persone del gruppo
  Persona* QualePersona(int i);
                // restituisce il puntatore alla i-esima
                // persona, con 1 <= i <= QuantePersone()
  // non interessa il resto
};
```

Si scriva una funzione esterna che, avendo come input un puntatore `r` ad un oggetto della classe `Ristorante` (che si sa avere tutti i tavoli vuoti), ed un vettore `g` di puntatori ad oggetti della classe `Gruppo`, assegni ogni persona rappresentata in `g` ad un tavolo di `r` in modo che persone dello stesso gruppo siano possibilmente sedute allo stesso tavolo.

Compito del 13 settembre 2001 **(soluzione a pagina 55)**

Si vuole progettare una classe in modo tale che ogni oggetto di essa rappresenti le informazioni riguardanti una biblioteca, i libri che possiede, ed i prestiti dei libri stessi. La classe deve mettere a disposizione le seguenti operazioni:

Acquisisce($l:$ *Libro*)**:** Il libro l viene acquisito dalla biblioteca.

Prestito($l:$ *Libro*, $p:$ *Persona*)**:** Il libro l viene dato in prestito alla persona p.
Precondizioni: La biblioteca possiede l e questo non è in prestito.

Restituzione($l:$ *Libro*)**:** Il libro l viene restituito.
Precondizioni: Il libro l é in prestito

$Verifica(l:$ *Libro*$) : (k:$ *intero*, $p:$ *Persona*)**:** L'intero k vale:

- 0 se il libro l non fa parte della biblioteca
- 1 se il libro l è presente in biblioteca
- 2 se il libro l è in prestito.

Se $k = 2$, allora p è la persona che ha il libro l in prestito, negli altri casi p non è significativo.

Si assumano già disponibili le classi `Persona` e `Libro` delle quali però non si conosce la definizione.

Esercizio 1 (punti 10) Si scriva la definizione della classe C++ `Biblioteca` con le funzioni che realizzano le operazioni suddette, oltre a tutti i dati, le altre funzioni e i costruttori che si ritengono opportuni.

Esercizio 2 (punti 12) Si scrivano le definizioni delle funzioni delle classi.

Esercizio 3 (punti 8) Si scriva una funzione esterna (non *friend*) che, avendo come parametri una biblioteca ed un vettore di libri, controlli lo stato dei libri del vettore. In particolare la funzione deve restituire:

- 0 se esiste almeno un libro che non fa parte della biblioteca;
- 1 se tutti i libri fanno parte della biblioteca, e nessun libro è in prestito;
- 2 se tutti i libri fanno parte della biblioteca b, e qualcuno di loro è in prestito.

Compito del 27 settembre 2001 (soluzione a pagina 59)

Si vuole realizzare una classe C++ che gestisca i tornei di *Pelota Rumena* (sport poco conosciuto simile al tennis). Ogni partita di Pelota Rumena si gioca al meglio dei tre set, cioè vince la partita il giocatore che vince due set. Il set è vinto dal giocatore che per primo raggiunge i 5 punti. Non esiste il "distacco di due", la partita può essere vinta anche 5-4.

Ogni oggetto della classe rappresenta un torneo. Non è rilevante la particolare formula di torneo (all'italiana, eliminazione diretta, ...), è però sicuro che ciascuna coppia di giocatori si può incontrare al massimo una volta.

La classe deve gestire sia le partite giocate, che quelle in corso, che quelle in programma ma non ancora cominciate. In particolare la classe deve offrire le seguenti operazioni.

AggiungiPartita($g1$: *Giocatore*, $g2$: *Giocatore*) Mette in programma la partita tra i giocatori $g1$ e $g2$.
Precondizione: non esiste una partita in corso o terminata tra $g1$ e $g2$.

IniziaPartita($g1$: *Giocatore*, $g2$: *Giocatore*) Inizia la partita tra $g1$ e $g2$. Il punteggio della partita è di 0-0 nel primo set.
Precondizione: la partita tra $g1$ e $g2$ è in programma.

SegnaPunto($g1$: *Giocatore*, $g2$: *Giocatore*) Si aggiunge un punto al giocatore $g1$ nel set corrente della sua partita con il giocatore $g2$ (ovviamente se il punto segnato determina la vittoria della partita o del set, questo evento deve essere registrato).
Precondizione: esiste una partita in corso tra $g1$ e $g2$.

AbbandonaPartita($g1$: *Giocatore*, $g2$: *Giocatore*) Il giocatore $g1$ si ritira dalla partita con $g2$. La partita è vinta da $g2$ indipendentemente dal punteggio corrente.
Precondizione: esiste una partita in corso tra $g1$ e $g2$.

StatoPartita($g1$: *Giocatore*, $g2$: *Giocatore*) : *intero* Restituisce lo stato della partita tra $g1$ e $g2$, nella codifica seguente:

- 0: terminata, ha vinto $g1$
- 1: terminata, ha vinto $g2$
- 2: in corso
- 3: in programma
- 4: inesistente (nessuna delle precedenti)

Esercizio 1 (punti 10) Si scriva la definizione della classe C++ `Torneo` e delle altre classi che si ritengono necessarie, considerando di riferirsi ad una classe `Giocatore` che si suppone già realizzata, della quale però non è nota la definizione.

Esercizio 2 (punti 16) Si scrivano le definizioni delle funzioni delle classi sopra definite.

Esercizio 3 (punti 4) Realizzare una funzione esterna (non *friend*) alla classe `Torneo` che prenda come parametri un giocatore g, un vettore di giocatori v ed un torneo t, e restituisca il numero di partite vinte da g in t contro i giocatori che sono elementi di v.

Compito del 10 dicembre 2001 (soluzione a pagina 65)

Si vuole realizzare una classe C++ che gestisca un'*agenzia matrimoniale*. L'agenzia ha un nome ed è descritta dalle seguenti operazioni:

InserisciCliente(p : *Persona*, s : *carattere*) una nuova persona p di sesso s diventa cliente dell'agenzia. Se la persona è già cliente, l'operazione non ha alcun effetto.

InserisciOpzione($p1$: *Persona*, $p2$: *Persona*) La persona $p1$ dichiara all'agenzia il suo interesse (opzione) a conoscere la persona $p2$. Ogni persona può opzionare solo una persona, quindi l'eventuale opzione precedente di $p1$ viene cancellata.
Precondizione: $p1$ e $p2$ sono entrambi clienti dell'agenzia e sono di sesso diverso.

CoppieCreate() : *vector*⟨*Persona*⟩ restituisce un vettore di persone contenente tutte le coppie che si sono opzionate a vicenda. Il vettore contiene nella locazione i (con $i = 0, 2, 4, \ldots$) la persona di sesso femminile e nella locazione $i + 1$ la persona di sesso maschile.

Esercizio 1 (punti 10) Si scriva la definizione della classe C++ `Agenzia` (e di eventuali classi ausiliarie che si ritengano opportune), considerando di riferirsi alla seguente classe `Persona` già realizzata e non modificabile.

```
class Persona
{public:
  Persona(string n, unsigned e) { nome = n; eta = e; }
  string Nome() const { return nome; }
  unsigned Eta() const { return eta; }
  bool operator==(const Persona& p) const
     { return nome == p.nome; }
  // ... altre funzioni che non interessano
 private:
  string nome;
  unsigned eta;
};
```

Esercizio 2 (punti 10) Si scrivano le definizioni delle funzioni della classe `Agenzia` (e delle eventuali classi ausiliarie definite nell'esercizio 1).

Esercizio 3 (punti 6) Realizzare una funzione esterna (non *friend* alla classe `Agenzia`) che prenda come parametro un vettore di agenzie e restituisca il numero totale di coppie create da tutte le agenzie in cui entrambi i componenti abbiano età superiore 50 anni.

Esercizio 4 (punti 4) Si discuta (a parole, eventualmente con qualche frammento di codice) una possibile soluzione alternativa degli esercizi 1 e 2 che non faccia uso della STL. In particolare, si discutano le differenze rispetto alla soluzione con la STL, evidenziando le funzioni che andrebbero riscritte ed eventuali funzioni che andrebbero aggiunte.

Compito del 7 gennaio 2002 **(soluzione a pagina 68)**

Un tabellone per il *Gioco dell'Oca* è costituito da una sequenza di caselle numerate da 1 a N. Ad ogni casella può essere associata una delle seguenti istruzioni:

- "Avanti di x caselle";
- "Indietro di y caselle";

dove x e y sono numeri interi positivi. Le caselle di partenza (la 1) e di arrivo (la N) non possono avere istruzioni associate.

Scopo del gioco è di arrivare con la propria pedina nella casella di arrivo (o di sorpassarla), a partire dalla casella di partenza, con una successione di mosse ognuna delle quali consiste di:

- lancio di un dado (a 6 facce);
- spostamento in avanti delle pedina di un numero di caselle pari al lancio del dado;
- esecuzione dell'eventuale istruzione associata alla casella di arrivo.

Si suppone che non possano esistere istruzioni che portino la pedina direttamente nella casella di arrivo o oltre, o che portino dietro alla casella di partenza. Si assume inoltre che l'esecuzione di una istruzione non porti mai ad una casella contenente un'altra istruzione.

Ad una partita del gioco partecipano K giocatori, il primo turno di lancio tocca al giocatore 1, poi il 2, fino al giocatore K e poi nuovamente al giocatore 1, e così via circolarmente. Vince chi per primo raggiunge la casella di arrivo.

Si vuole realizzare una classe C++ che gestisca le partite del gioco dell'oca. Le operazioni della classe sono le seguenti:

Ricomincia() Le pedine di tutti i giocatori vengono portate nella casella di partenza. Il turno tocca al giocatore 1.

DefinisciTabellone($t : vector\langle intero\rangle$) Il tabellone viene sostituito con quello descritto da t. Il vettore t descrive il tabellone nel seguente modo: se $t[i] = 0$ la casella i è normale; se $t[i] > 0$ la casella i è del tipo "avanti di x caselle" con $x = t[i]$; se $t[i] < 0$ la casella i è del tipo "indietro di y caselle" con $y = -t[i]$. La partita viene ricominciata.

DefinisciGiocatori($k : intero$) Il numero di giocatori diventa k. La partita viene ricominciata.

EseguiMossa($d : intero$) Il giocatore di turno esegue la mossa corrispondente al lancio del dado pari a d (con $1 \leq d \leq 6$). Il turno passa al giocatore successivo.
Precondizione: la partita non è ancora stata vinta da alcun giocatore.

Vincitore() : *intero* Restituisce 0 se la partita è ancora in corso, altrimenti restituisce il giocatore che ha vinto.

Esercizio 1 (punti 10) Si scriva la definizione della classe C++ `GiocoOca` (e di eventuali classi ausiliarie che si ritengano opportune).

Esercizio 2 (punti 10) Si scrivano le definizioni delle funzioni della classe `GiocoOca` (e delle eventuali classi ausiliarie definite nell'esercizio 1).

Esercizio 3 (punti 6) Si scriva una funzione esterna che prenda come parametri:

- un tabellone (descritto tramite vettore di interi)
- un numero di giocatori
- un vettore che rappresenta la sequenza dei lanci

e restituisca il vincitore della partita ottenuta eseguendo la sequenza di lanci data.

Si assuma che la sequenza del lanci sia sufficiente per finire la partita. La sequenza dei lanci non deve necessariamente essere usata per intero.

Esercizio 4 (punti 4) Si scrivano gli operatori "`<<`" e "`>>`" per la classe `GiocoOca`, che leggano e scrivano lo stato di una partita nel formato che si ritiene più opportuno.

Compito del 8 aprile 2002 (soluzione a pagina 71)

Per giocare agli *scacchi nibelunghi* si utilizza una scacchiera 5×5. Ognuno dei due giocatori ha inizialmente 5 pedine ed esiste inoltre un pezzo neutro, chiamato *Zen*. I pezzi vengono inizialmente piazzati come in figura, dove il simbolo ∘ denota le pedine bianche, il simbolo • denota le pedine nere e Z denota lo Zen. Il primo giocatore a muovere è il bianco.

∘	∘	∘	∘	∘
		Z		
•	•	•	•	•

Le pedine si muovono di una sola casella in orizzontale o verticale, avanti o indietro, mentre lo Zen si muove sempre di una sola casella ma anche in diagonale (cioè come il re degli scacchi tradizionali).

Ad ogni turno un giocatore può spostare a scelta una sua pedina oppure lo Zen. Se una pedina propria si sposta su una pedina avversaria, quest'ultima viene mangiata e si elimina dal gioco. Se una pedina propria si sposta sullo Zen il giocatore vince e la partita è finita. Non è consentito spostare una pedina su un'altra propria o spostare lo Zen su una qualsiasi pedina (propria o dell'avversario).

Esercizio 1 (punti 10) Si scriva la definizione di una classe C++ `Partita` che permetta di eseguire le mosse del gioco e di ispezionare la situazione corrente. La classe deve inoltre permettere di interrompere

una partita e di tornare alla posizione iniziale. La classe non deve però consentire di effettuare mosse non ammissibili.

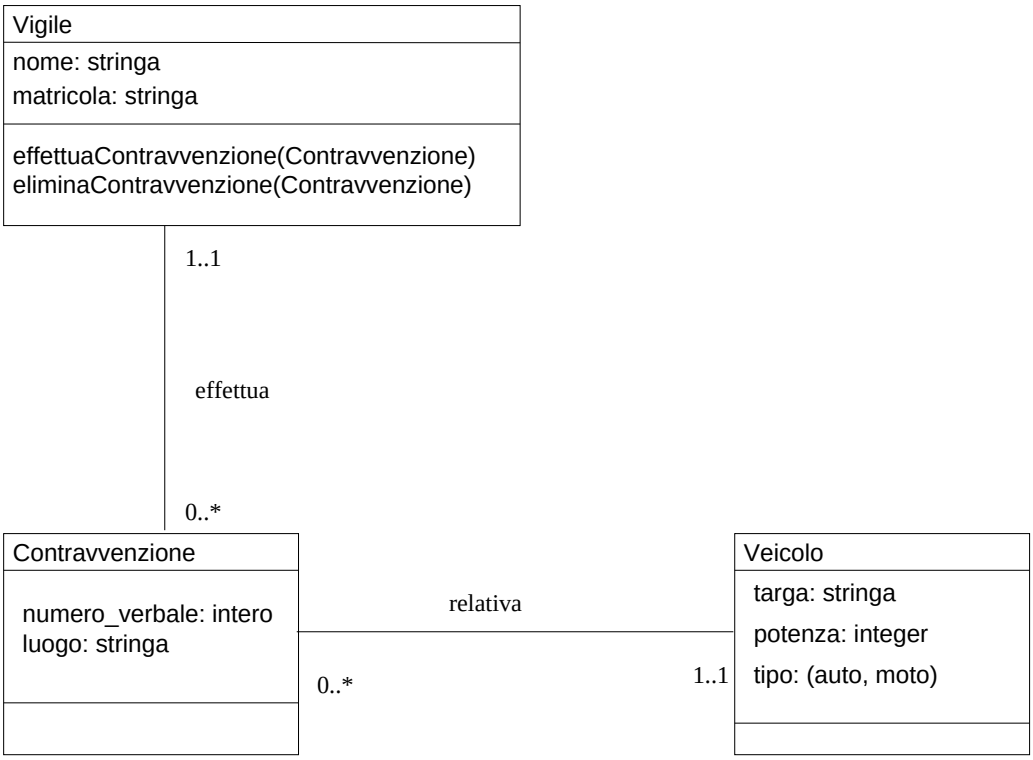
Esercizio 2 (punti 10) Si scrivano le definizioni delle funzioni della classe `Partita`.

Esercizio 3 (punti 5) Realizzare una funzione esterna (non *friend*) alla classe `Partita` che prenda come parametro una partita e verifichi se il giocatore di turno può o meno vincere la partita con una mossa.

Esercizio 4 (punti 5) Realizzare una funzione esterna (non *friend*) alla classe `Partita` che prenda come parametri una partita ed una mossa (rappresentate come si ritiene opportuno) ed esegua la mossa sulla partita. La funzione deve restituire un valore che indichi se il giocatore che ha eseguito la mossa ha vinto la partita oppure no.

Compito del 25 giugno 2002 **(soluzione a pagina 76)**

L'applicazione da progettare riguarda le informazioni sulle contravvenzioni elevate in un comune, come descritte nel seguente diagramma UML.



Esercizio 1 (punti 10) Si scrivano le definizioni di tre classi C++ `Vigile`, `Contravvenzione` e `Veicolo` corrispondenti alle classi UML omonime. Si dotino le classi di tutte le funzioni di ispezione dei dati che si ritengono opportune (vedere anche esercizi 3 e 4).

Esercizio 2 (punti 10) Si scrivano le definizioni delle funzioni della classe `Vigile` e di almeno una delle altre due classi.

Esercizio 3 (punti 5) Si scriva una funzione esterna (non *friend*) booleana che, dato un vigile, verifichi se ha elevato una contravvenzione per più di una volta ad uno stesso veicolo.

Esercizio 4 (punti 5) Si scriva una funzione esterna (non *friend*) booleana che, dato un vigile, verifichi se ha elevato più contravvenzioni alle moto di quante ne abbia elevate alle automobili.

Compito del 15 luglio 2002 (soluzione a pagina 80)

L'applicazione da progettare riguarda la gestione delle prenotazioni di un singolo concerto in un teatro. Il teatro ha un numero fissato di posti in platea ed in galleria, e i posti *non sono numerati*. I prezzi per la platea e per la galleria sono anch'essi fissati (niente riduzioni) e diversi tra loro. Per semplicità si suppongano che i prezzi siano in Euro senza centesimi.

Ogni prenotazione riguarda un singolo posto, ed una persona può mantenere attiva una sola prenotazione. Le operazioni che si possono eseguire sono le seguenti:

InserisciPrenotazione($p : Persona,\ t : boolean,\ c : boolean) : intero$ Viene inserita una prenotazione per la persona p, in platea se $t = true$, in galleria se $t = false$. Se $c = true$ la prenotazione viene registrata come "flessibile", cioè la persona è disposta anche ad accettare un posto nell'altro settore in caso non ci sia disponibilità in quello richiesto.

La prenotazione viene memorizzata indipendentemente dalla disponibilità. Una prenotazione non soddisfacibile viene messa in attesa, e può essere soddisfatta a seguito di eventuali eliminazioni.

Non è possibile modificare una prenotazione tramite questa funzione. Se p è già presente nel sistema, la funzione termina senza eseguire modifiche.

La funzione restituisce:

- `0` se la persona è già prenotata
- `1` se la prenotazione è soddisfacibile senza cambi
- `2` se la prenotazione è soddisfacibile ma cambiando settore (possibile solo se $c = true$)
- `3` se la prenotazione non è attualmente soddisfacibile.

EliminaPrenotazione($p : Persona$) La prenotazione della persona p viene eliminata. Al momento dell'eliminazione di una prenotazione, lo stato della prima (in ordine di inserimento) tra le prenotazioni in attesa oppure sottoposte a cambio di settore viene modificato. Se viene modificata una prenotazione con cambio di settore, allora è possibile che un'ulteriore prenotazione in attesa per tale settore venga cambiata di stato.

Esercizio 1 (punti 10) Si scriva la definizioni della classe C++ `Concerto`. Si doti la classe di tutte le funzioni di ispezione dei dati che si ritengono opportune (vedere anche esercizi 3 e 4).

Si utilizzi la classe `Persona` che si suppone già realizzata, della cui definizione è nota soltanto l'esistenza dell'operatore "`==`" che verifica se due oggetti rappresentano la stessa persona.

Esercizio 2 (punti 10) Si scrivano le definizioni delle funzioni della classe `Concerto`.

Esercizio 3 (punti 5) Si scriva una funzione esterna che, dato un oggetto della classe `Concerto`, restituisca l'incasso totale che si ottiene assumendo che i biglietti venduti siano esattamente quelli corrispondenti alle prenotazioni correntemente soddisfatte.

Esercizio 4 (punti 5) Si scriva una funzione esterna che riceva come parametri un oggetto della classe `Concerto` ed un vettore di prenotazioni (rappresentate nel modo che si ritiene più opportuno), e inserisca le prenotazioni nel sistema in modo ordinato (partendo dalla locazione 0 del vettore).

L'insieme di prenotazioni può anche contenere *riprenotazioni* della stessa persona, in tal caso la funzione deve prima eliminare le prenotazione vecchia, e poi inserire quella nuova. È inteso che una persona che modifica la sua prenotazione potrebbe perdere il posto a favore di persone che in quel momento sono in attesa.

Compito del 9 settembre 2002 (soluzione a pagina 87)

Si vuole progettare una classe in modo tale che ogni oggetto rappresenti la forza lavoro di un'azienda e l'orario di lavoro giornaliero degli impiegati. Ogni azienda ha un massimo di impiegati che può assumere; questo numero è fisso, e non può essere modificato. Il lavoro è organizzato in turni, determinati dall'orario di inizio e da quello di fine turno. Un impiegato può lavorare anche più di un turno, a patto che questi non si sovrappongano. Ad esempio un impiegato può lavorare contemporaneamente nei tre turni 8-12, 14-17 e 17-20, ma non nei due turni 9-13 e 11-16. Le operazioni principali dell'applicazione sono:

Assumi(i : *Impiegato*) : L'impiegato i viene assunto dall'azienda. Se i era già impiegato nell'azienda, allora l'operazione non ha effetti.
Precondizione: Il numero di impiegati dell'azienda è inferiore al massimo.

CreaTurno(t : *Turno*) Viene inserito il turno di lavoro t. Nessun impiegato è inizialmente assegnato a questo turno.
Precondizione: Non esiste già il turno t, cioè un turno che inizia e finisce alla stessa ora di t.

AssegnaTurno(i : *Impiegato*,t : *Turno*) Il turno t viene aggiunto a quelli assegnati all'impiegato i.
Precondizioni:

- L'impiegato i è stato assunto dalla ditta,
- il turno t esiste e non è già assegnato all'impiegato i.
- l'impiegato i non è già assegnato ad un turno che si sovrapponga temporalmente con t.

Esercizio 1 (punti 10) Assumendo disponibili, già realizzate, le classi `Impiegato` e `Turno` e la funzione `Disgiunti()` definite in seguito, si scriva la definizione della classe C++ `Azienda` e di eventuali altre classi che risultino necessarie. Si doti la classe `Azienda` di tutte le funzioni di ispezione dei dati che si ritengono opportune (vedere anche esercizi 3 e 4).

Le definizioni delle classi `Impiegato` e `Turno` e della funzione `Disgiunti()` sono le seguenti:

```
class Impiegato
{
 public:
  Impiegato(string n) { nome = n; }
  string Nome() const { return nome; }
  bool operator==(const Impiegato& p)
    const { return nome == p.nome; }
 private:
  string nome;
};
```

```
class Turno
{
 public:
  Turno(int s1, int s2)
  { assert (s1 < s2); start = s1; stop = s2; }
  int Start() const { return start; }
  int Stop() const { return stop; }
  int NumOre() const { return stop - start; }
 private:
  int start, stop;
};
```

```
bool Disgiunti(Turno* t1, Turno* t2)
// restituisce true se i turni puntati da t1 e t2 non si sovrappongono (es. 9-12 e 15-19)
// restituisce false altrimenti (es. 9-15 e 12-18)
{ return t1->Stop() < t2->Start() || t2->Stop() < t1->Start(); }
```

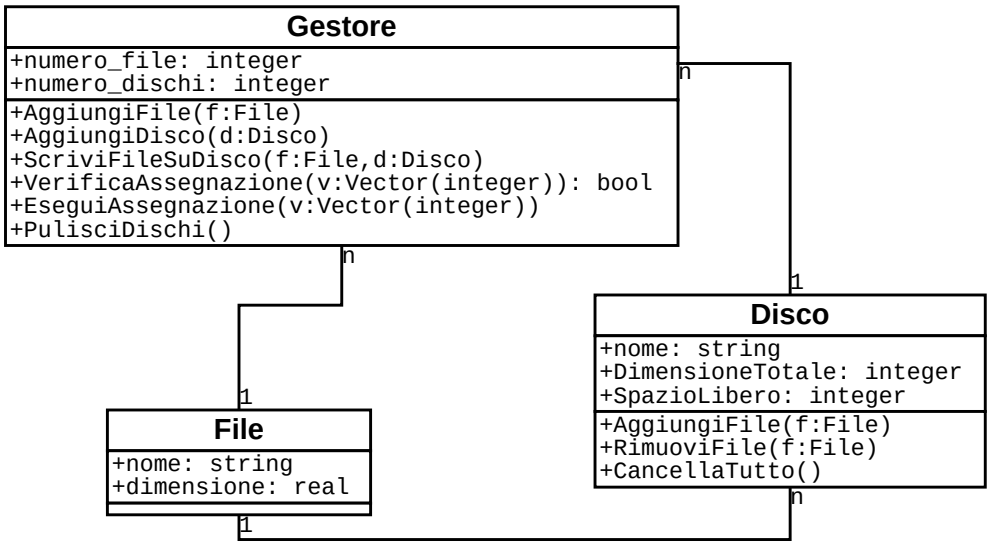
Esercizio 2 (punti 12) Si scrivano le definizioni delle funzioni della classe `Azienda` e delle eventuali altri classi definite.

Esercizio 3 (punti 4) Si scriva una funzione esterna che riceva come parametri un oggetto della classe `Azienda` e due interi che denotano l'ora di inizio e di fine di un turno, e restituisca `true` se il turno è presente nell'azienda, `false` altrimenti.

Esercizio 4 (punti 4) Si scriva una funzione esterna che, dato un oggetto della classe `Azienda`, restituisca il numero totale di ore lavorate dall'insieme dei suoi impiegati.

Compito del 25 settembre 2002 **(soluzione a pagina 92)**

Il gestore di un sistema informatico ha a disposizione vari dischi magnetici, ognuno con una propria capacità espressa in gigabyte (ad esempio, 13.1 GB), su cui immagazzinare i sui file. Il gestore deve occuparsi di effettuare l'allocazione dei file sui dischi, tenendo conto che i file sono indivisibili. Il diagramma UML dell'applicazione appena descritta è il seguente:



Le operazioni *AggiungiFile* e *AggiungiDisco* aggiungono rispettivamente un nuovo file e un nuovo disco al sistema, senza fare alcuna assegnazione.

L'operazione *EseguiAssegnazione* esegue l'assegnazione dei file ai dischi in base al contenuto del vettore v, che ha lunghezza pari al numero di file del sistema. Il file i-esimo inserito nel sistema deve essere allocato nel dischi j-esimo, dove $j = v[i]$. La funzione deve preventivamente disfare qualsiasi assegnazione precedente dei file ai dischi. La precondizione dell'operazione è che ci sia spazio sufficiente su tutti i dischi.

L'operazione *ScriviFileSuDisco* invece scrive un dato file su un dato disco, con le precondizioni che sia il file che il disco appartengano al sistema e che ci sia spazio sufficiente sul disco.

L'operazione *VerificaAssegnazione* controlla che un'assegnazione, passata tramite vettore di interi come per l'operazione precedente, è fattibile senza superare la capacità di alcun disco. L'operazione non deve effettuare alcuna assegnazione, ma sono verificarne la fattibilità. Come precondizione, si assuma che nessun file è assegnato ad alcun disco.

L'operazione *PulisciDischi* elimina tutte le assegnazioni dei file sui dischi.

Esercizio 1 (punti 10) Assumendo disponibile, già realizzata, la classe `File`, si scriva la definizione delle classi C++ `Gestore` e `Disco`. La definizione della classe `File` è la seguente:

```
class File
{public:
  File(string n, float c) { nome = n; dimensione = c; }
  string Nome() const { return nome; }
  float Dimensione() const { return dimensione; }
  bool operator==(const File& f) const
    { return nome == f.nome; }
 private:
  string nome;
  float dimensione;
};
```

Esercizio 2 (punti 10) Si scrivano le definizioni delle funzioni delle classi.

Esercizio 3 (punti 6) Si scriva una funzione esterna che riceva come parametri un vettore di file e due dischi e restituisca tramite puntatore un oggetto della classe `Gestore` che abbia quanti più file possibile scritti sul primo disco ed i rimanenti sul secondo (che si suppone di dimensione sufficientemente grande). Si assuma che i file nel vettore siano ordinati in senso crescente di dimensione.

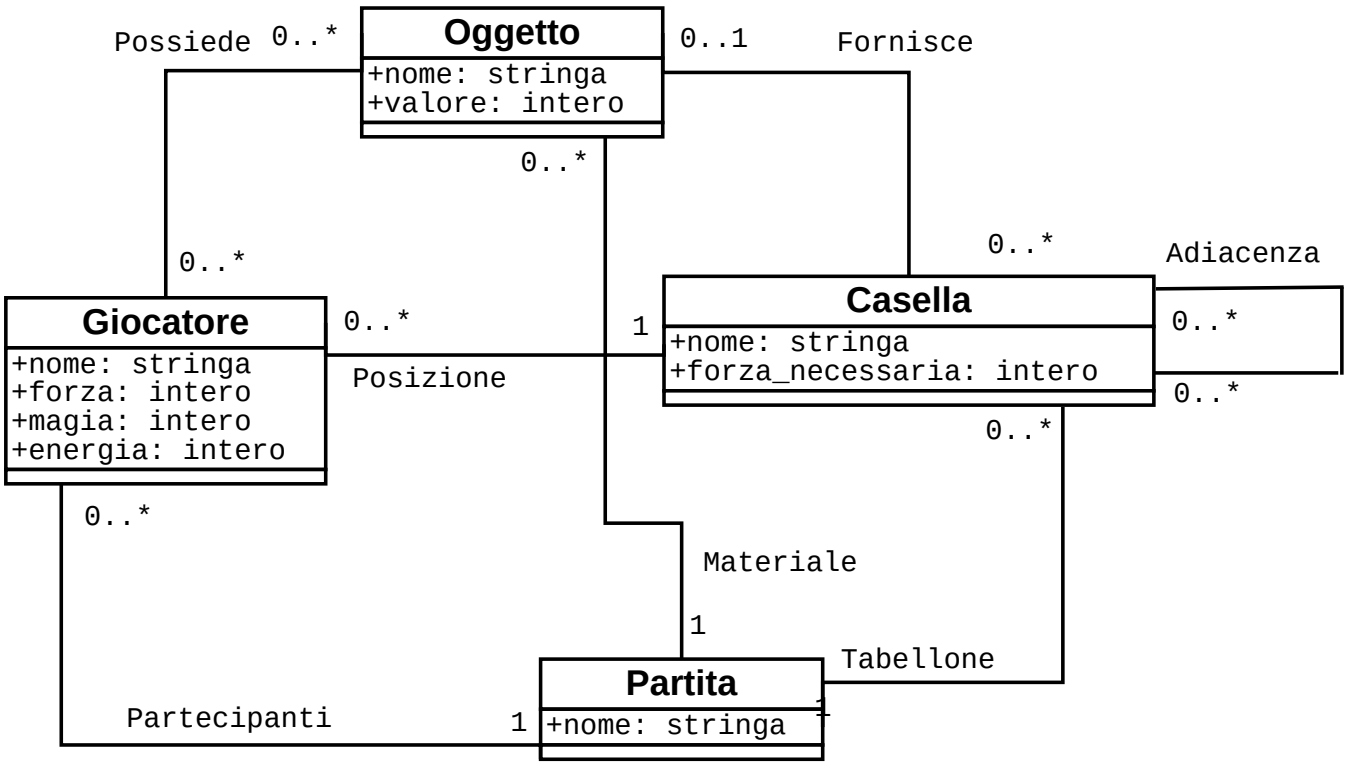
Esercizio 4 (punti 4) Si scriva l'operatore `<<` della classe `Gestore`.

Compito del 5 dicembre 2002 [3 ore di tempo] **(soluzione a pagina 98)**

Si vuole progettare una classe per la gestione di partite di un *gioco di ruolo*. Una partita è rappresentata da un insieme di giocatori, un insieme di caselle su cui transitano i giocatori, ed un insieme di oggetti che i giocatori raccolgono dalle caselle che li forniscono. La possibilità di muoversi da una casella all'altra

è denotata da una relazione di *adiacenza* tra caselle che viene definita dinamicamente (e non fissata a priori come in una scacchiera).

L'applicazione è descritta dal seguente schema in UML, senza operazioni. Le operazioni della classe Partita sono specificate di seguito, mentre le altre classi sono dotate solo di semplici operazioni di inserimento e modifica di singoli dati, che sono state omesse.



Nello schema UML l'attributo forza_necessaria della classe Casella rappresenta la forza necessaria ad un giocatore per poter prendere l'oggetto presente nella casella. Si noti che una casella può avere al massimo un oggetto presente. Inoltre, quando un oggetto viene preso in una casella, questo comunque rimane ancora disponibile nella casella per gli altri giocatori (cioè si assume che un oggetto sia fisicamente disponibile in una casella in un numero illimitato di copie).

Gli attributi forza e magia del giocatore sono delle costanti, mentre energia è un dato variabile che parte dal valore iniziale di 10 e evolve durante il gioco.

Si assuma infine che i nomi dei giocatori, delle caselle e degli oggetti siano sempre univoci.

Le funzioni della classe Partita sono specificate come segue:

AggiungiGiocatore$(g : Giocatore, c : Casella)$: g viene aggiunto alla partita e viene posizionato sulla casella c.
Precondizioni: g non è già un partecipante della partita, c è una casella del tabellone.

AggiungiCasella$(c : Casella, v : vettore\langle Casella\rangle)$: c viene aggiunta al tabellone della partita e resa adiacente a tutte la caselle di v.
Precondizioni: c non è già nel tabellone, tutte le caselle di v sono caselle del tabellone.

AggiungiOggetto$(o : Oggetto, c : Casella)$: o viene aggiunto alla partita e posizionato nella casella c.
Precondizioni: o non è già nella partita, c è una casella della partita.

MuoviGiocatore$(g : Giocatore, c : Casella)$: g viene spostato nella casella c se questa è adiacente alla sua posizione corrente, altrimenti g rimane al suo posto.
Precondizioni: g e c sono elementi della partita.

PrendiOggetto$(g : Giocatore)$: g prende l'oggetto presente nella sua posizione corrente, l'oggetto rimane comunque disponibile nella casella.
Precondizioni: g è un partecipante della partita ed ha la forza necessaria per prendere l'oggetto in quella casella.

Attacca$(g1 : Giocatore, g2 : Giocatore)$: $g1$ esegue un attacco su $g2$. Vince il giocatore che ottiene il punteggio più alto come somma della sua forza più il lancio di un dado (in caso di parità vince il giocatore attaccato). L'energia del perdente viene decrementata di 1 e tutti i suoi oggetti passano al vincitore.
Precondizioni: $g1$ e $g2$ sono partecipanti della partita e si trovano nella stessa casella.

Esercizio 1 (punti 10) Si scrivano le definizioni delle classi C++ `Oggetto`, `Casella`, `Giocatore` e `Partita`.

Esercizio 2 (punti 10) Si scrivano le definizioni delle funzioni delle quattro classi. Si assuma disponibile la funzione `int Random(int a,int b)` che restituisce un numero casuale tra `a` e `b`, estremi inclusi (con `a <= b`).

Esercizio 3 (punti 6) Si scriva una funzione esterna che prenda un parametro di tipo partita e lo modifichi rendendo tutte le sue caselle adiacenti tra loro.

Esercizio 4 (punti 4) Si considerino le seguenti definizioni delle classi `A` e `B`

```
class A
{public:
  A(int g) { x = g; }
  int X() const { return x; }
  void I() { x++; }
private:
  int x;
};
```

```
class B
{public:
  B(int u) { p_a = new A(u); }
  ~B() { delete p_a; }
  int S() const { return p_a->X(); }
  void C() { p_a->I(); }
private:
  A* p_a;
};
```

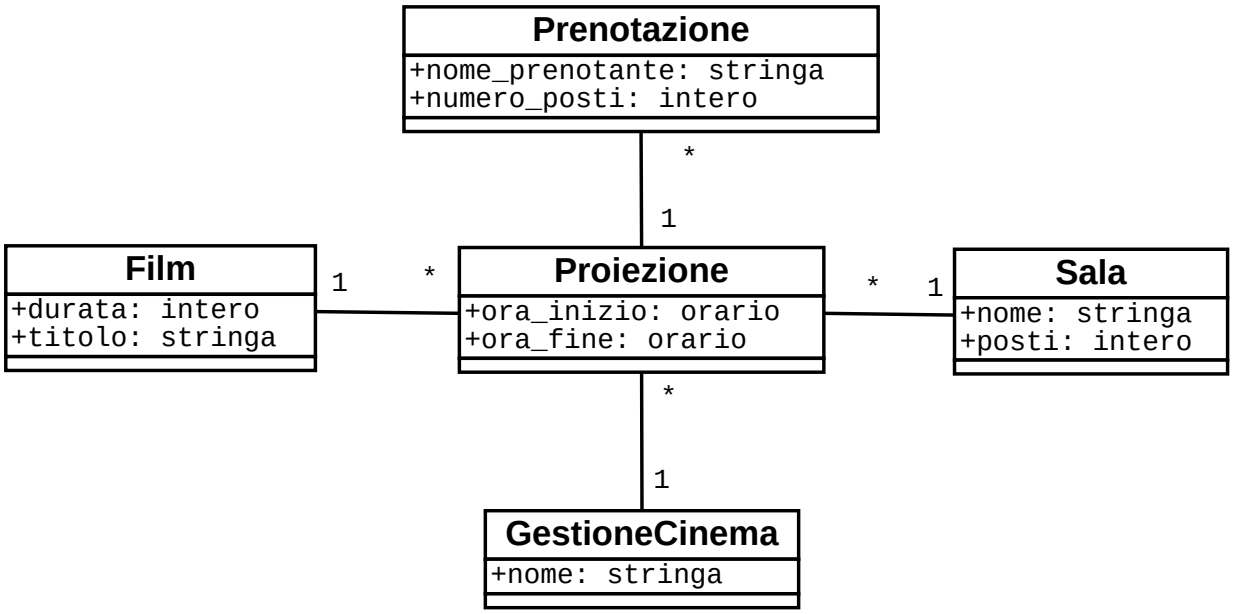
- Dire che cosa stampa la funzione `main()` seguente (motivando la risposta), in cui `m` è inizializzato con il vostro numero di matricola
- Si scriva il costruttore di copia della classe `B` in modo da non dar luogo a condivisione di memoria

```
int main()
{
  int m = <matricola>;
  B b1(m % 10);
  B b2 = b1;
  b2.C();
  cout << b1.S() << endl;
}
```

Compito del 18 dicembre 2002 **(soluzione a pagina 105)**

Si vuole progettare una classe per la gestione del sistema di prenotazione per le proiezioni giornaliere di un cinema multisala. Una proiezione è caratterizzata dal film proiettato, la sala in cui è proiettato ed un insieme di prenotazioni ad essa associate.

L'applicazione è descritta dal seguente schema in UML, senza operazioni. Le operazioni della classe GestioneCinema sono specificate di seguito; le classi Sala, Cinema e Prenotazione non hanno operazioni; le semplici operazioni della classe Proiezione sono state omesse.



Le funzioni della classe GestioneCinema sono specificate come segue:

$InserisciProiezione(p : Proiezione)$: p viene aggiunta alla programmazione.
Precondizioni: la sala di p è libera da altre proiezioni per tutto l'orario di p; la proiezione p non ha prenotazioni.

$InserisciPrenotazione(pn : Prenotazione, p : Proiezione) : booleano$ se il numero di posti già prenotati per p sommati ai posti di pn non supera la capienza delle sala di p, allora pn viene aggiunta alle prenotazioni di p e viene restituito $true$, altrimenti la prenotazione non viene inserita e viene restituito $false$.
Precondizioni: p è una proiezione già presente nel cinema.

$CambiaOrario(p : Proiezione, i : Orario, f : Orario)$: l'orario di p viene cambiato in $i \div f$ (ora_inizio: i, ora_fine: f)
Precondizioni: p è una proiezione già presente; la sala di p è libera per tutto l'orario $i \div f$.

$CambiaSala(p : Proiezione, s : Sala)$: la sala di p viene cambiata in s
Precondizioni: p è una proiezione già presente; la sala s ha un numero di posti sufficiente per le prenotazioni correnti di p; la sala s è libera per l'orario di p.

Esercizio 1 (punti 10) Si scrivano le definizioni delle classi C++ `Proiezione` e `GestioneCinema` sulla base delle definizioni date delle classi `Film`, `Sala`, `Prenotazione` e `Orario`, in cui alcune definizioni di funzione sono omesse per brevità (ma possono essere assunte già realizzate).

Esercizio 2 (punti 10) Si scrivano le definizioni delle funzioni delle due classi gestendo le precondizioni attraverso la funzione `assert()`.

Esercizio 3 (punti 4) Si scriva una funzione esterna che prenda come parametro un oggetto di tipo `GestioneCinema` e restituisca *true* se esiste un film che abbia durata maggiore della lunghezza della sua proiezione, $false$ altrimenti. A questo scopo si utilizzi l'operatore "`-`" della classe `Orario` che restituisce la differenza in minuti tra due orari.

Esercizio 4 (punti 6) Si scriva una funzione esterna che prenda come parametri un oggetto di tipo `GestioneCinema`, un vettore di nomi di persone, ed un intero k e restituisca il numero di persone del vettore che hanno fatto delle prenotazioni nel cinema per un numero complessivo di posti pari o superiore a k.

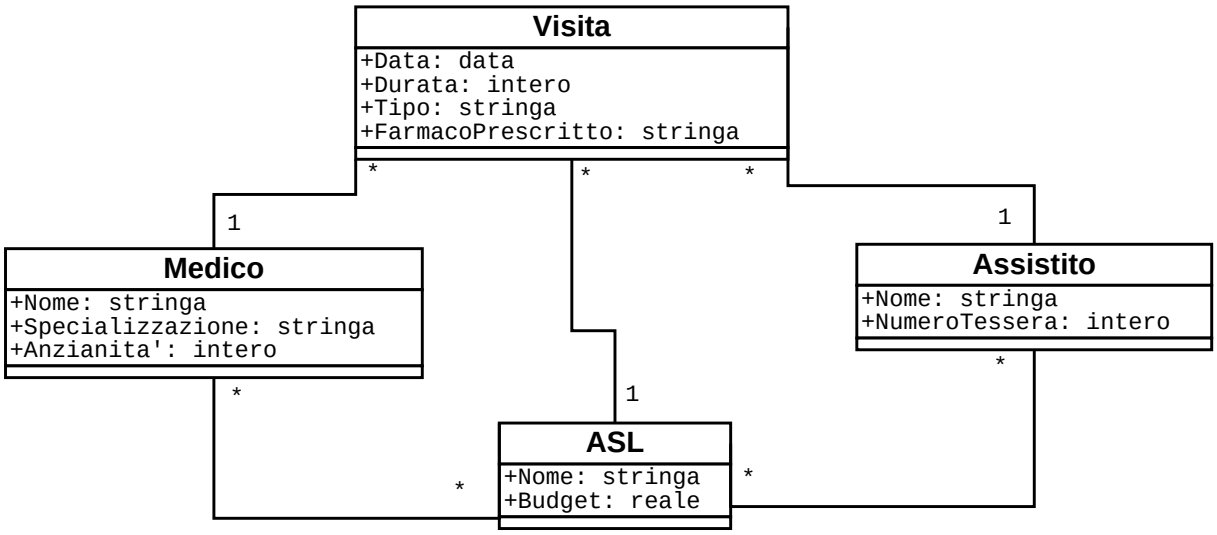
```
class Orario
{ friend bool operator<(const Orario& o1, const Orario& o2);
  friend bool operator<=(const Orario& o1, const Orario& o2);
  friend bool operator==(const Orario& o1, const Orario& o2);
  friend int operator-(const Orario& o1, const Orario& o2);  // differenza in minuti
public:
  Orario(int o, int m);
  int Ore() const { return ore; }
```

```
  int Minuti() const { return minuti; }
private:
  int ore, minuti;
};

class Film
{ friend bool operator==(const Film& f1, const Film& f2);
public:
  Film(string t, int d);
  int Durata() const { return durata; }
  string Titolo() const { return titolo; }
private:
  string titolo;
  int durata; // in minuti
};

class Sala
{ friend bool operator==(const Sala& s1, const Sala& s2);
public:
  Sala(string n, int c);
  int Capienza() const { return capienza; }
  string Nome() const { return nome; }
private:
  string nome;
  int capienza;
};

class Prenotazione
{public:
  Prenotazione(string n, int posti);
  string NomePrenotante() const { return nome_prenotante; }
  int Posti() const { return posti; }
private:
  string nome_prenotante;
  int posti;
};
```

Compito del 7 luglio 2003 (soluzione a pagina 112)

Si vuole progettare una classe per la gestione dei servizi gestiti da una ASL. Ciascuna ASL è caratterizzata da un insieme di medici, un insieme di assistiti, e le visite effettuate dai medici agli assistiti.

L'applicazione è descritta dal seguente schema in UML, senza operazioni. Le operazioni della classe ASL sono specificate di seguito; le classi Medico e Assistito non hanno operazioni; le semplici operazioni della classe Visita sono state omesse.



Le funzioni della classe ASL sono specificate come segue:

$InserisciAssistito(a : Assistito)$: L'assistito a viene inserito nella ASL.
Precondizioni: il numero di tessera di a non è già presente nella ASL.

$InserisciMedico(m : Medico)$: Il medico m viene inserito nella ASL. Se m è già presente allora l'operazione non ha effetto.

$InserisciVisita(v : Visita)$: La visita v viene inserita tra le visite effettuate.
Precondizioni: il medico ed il paziente di v sono già presente nella ASL. Non è già presente una visita dello stesso medico allo stesso paziente nella stessa data e dello stesso tipo.

$CreaVisita(p : Medico,\ a : Assistito,\ d : Data,\ f : stringa)$: Viene inserita una visita di p ad a di durata 30' (durata standard), di tipo uguale alla specializzazione di p, in cui viene prescritto il farmaco f.
Precondizioni: non è già presente una visita dello stesso medico allo stesso paziente nella stessa data e dello stesso tipo.

$CorreggiSpecializzazione(m : Medico,\ s : stringa)$: la specializzazione di m viene cambiata in s. Per tutte le visite effettuate da m che hanno il tipo uguale alla specializzazione di m, questo viene cambiato in s.
Precondizioni: m è un medico già presente; la sua specializzazione è diversa da s.

Si assuma che non possano esistere medici che abbiano lo stesso nome e che non possano esistere assistiti che abbiano lo stesso numero di tessera.

Esercizio 1 (punti 10) Si scrivano le definizioni delle classi C++ `Asl` e `Visita` avendo già disponibili le classi `Medico`, `Assistito` e `Data`. Le definizioni delle classi `Medico` e `Assistito` sono quelle fornite di seguito, in cui alcune definizioni di funzione sono omesse per brevità (ma possono essere assunte già realizzate).
Si consideri disponibile la classe `Data` con tutte le funzioni di gestione che si ritengono necessarie.

Esercizio 2 (punti 10) Si scrivano le definizioni delle funzioni delle due classi, gestendo le precondizioni attraverso la funzione `assert()`.

Esercizio 3 (punti 6) Si scriva una funzione esterna che prenda come parametro una ASL e restituisca il nome del medico della ASL che ha il massimo rapporto tra il numero di visite di tipo corrispondente alla sua specializzazione ed il numero di visite totali che ha effettuato.

Esercizio 4 (punti 4) Si scriva l'operatore "`<<`" per la classe `Asl` in modo che stampi tutte le informazioni ad essa associate.

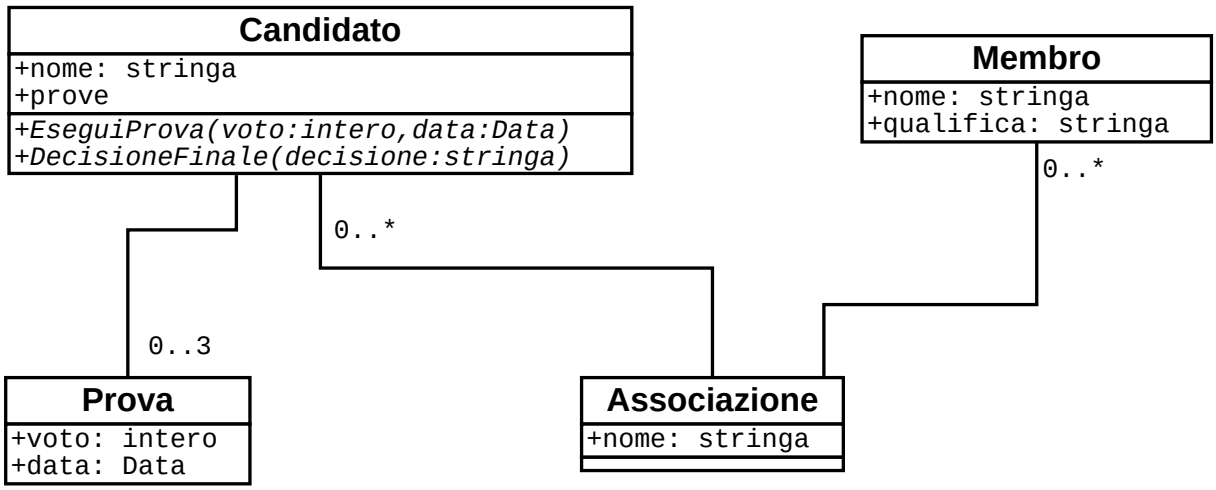
Definizione delle classi già disponibili:

```
class Medico
{
  friend ostream& operator<<(ostream& os, const Medico& m);
public:
  Medico(string n, string s, int a);
```

```
  string Nome() const {return nome; }
  string Specializzazione() const {return spec; }
  void CambiaSpecializzazione(string s) { spec = s; }
  int Anzianita() const { return anzianita; }
private:
  string nome, spec;
  int anzianita;
};

class Assistito
{
  friend ostream& operator<<(ostream& os, const Assistito& a);
public:
  Assistito(string n, int nt);
  string Nome() const {return nome; }
  int NumTessera() const { return num_tessera; }
private:
  string nome;
  int num_tessera;
};
```

Compito del 2 settembre 2003 (soluzione a pagina 117)

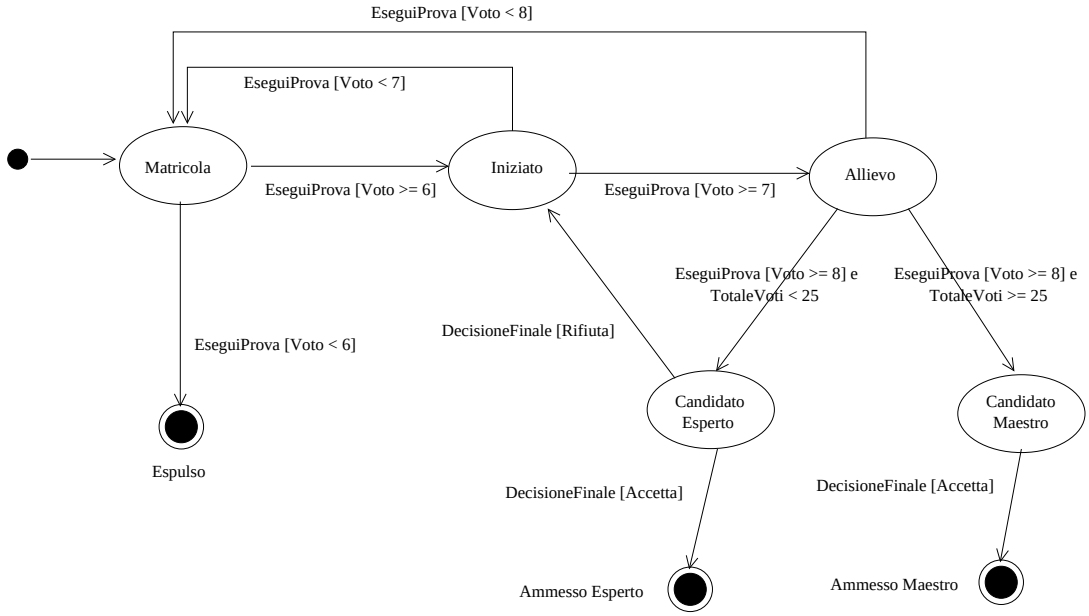
Si vuole progettare una classe per la gestione degli iscritti di alcune associazioni. Ciascuna associazione ha un insieme di affiliati che si dividono in *membri* e *candidati*, come descritto dal seguente diagramma UML.



Come si evince dal diagramma, dei membri interessa solo il nome e la qualifica (*maestro* o *esperto*). Riguardo ai candidati, questi per diventare membri devono superare tre prove, ciascuna delle quali attribuisce un voto da 1 a 10. Il voto minimo per superare una prova varia da prova a prova.

I candidati appena iscritti vengono chiamati Matricole, quelli che hanno superato la prima prova o le prime due prove vengono chiamati Iniziati e Allievi rispettivamente. I candidati che hanno superato tutte e tre le prove diventano CandidatiMaestro oppure CandidatiEsperto a seconda del punteggio totale ottenuto nelle tre prove.

Il percorso preciso per diventare membri (maestri o esperti) oppure per essere espulsi è descritto dal seguente diagramma degli stati, che specifica le funzioni della classe Candidato.



Le funzioni della classe Associazione sono specificate come segue:

InserisciCandidato($n : stringa$) : Viene inserito (come ultimo) nell'associazione un nuovo candidato Matricola il cui nome è n.
Precondizioni: Non esiste già un affiliato di nome n.

EseguiProve($d : Data, v : vettore\langle intero\rangle$) : Tutti i candidati negli stati Matricola, Iniziato o Allievo eseguono l'operazione EseguiProva in data d. Il vettore v ha un numero di locazioni pari al numero di candidati in questi stati, e contiene i loro voti in ordine di ingresso nell'associazione.

RegistraAccettazioni($v : vettore\langle boolean\rangle$) : Vengono eseguite le transizioni dagli stati `CandidatoMaestro` e CandidatoEsperto. Il vettore v ha un numero di locazioni pari al numero di candidati nello stato CandidatoEsperto e memorizza le loro decisioni (vero = accetta, falso = rifiuta)

NominaNuoviMembri() : Tutti i candidati negli stati AmmessoEsperto e Ammessomaestro vengono nominati membri, tutti i candidati nello stato Espulso vengono eliminati

Esercizio 1 (punti 10) Si scrivano le definizioni delle classi C++ `Candidato` e `Associazione` avendo già disponibili le classi `Prova`, `Membro` e `Data`. Le definizioni delle classi `Prova` e `Membro` sono quelle fornite di seguito, in cui alcune definizioni di funzione sono omesse per brevità (ma possono essere assunte già realizzate). Si consideri disponibile la classe `Data` con tutte le funzioni di gestione che si ritengono necessarie.

Esercizio 2 (punti 15) Si scrivano le definizioni delle funzioni delle due classi, gestendo le precondizioni attraverso la funzione `assert()`.

Esercizio 3 (punti 5) Si scriva un *driver* per verificare la correttezza dell'implementazione della classe `Associazione`.

Definizione delle classi disponibili:

```
class Membro
{
 public:
  Membro(string n, string q);
  string Nome() const { return nome; }
  string Qualifica() const { return qualifica; }
 private:
  string nome;
  string qualifica;
};
```

```
class Prova
{
 public:
  Prova(unsigned v, Data d);
  unsigned voto;
  Data data;
};
```

Compito del 16 settembre 2003 (soluzione a pagina 124)

Si richiede di progettare una classe `CentroVaccinazioni` per la gestione di un *centro di vaccinazioni*. Ogni vaccino è identificato da un codice, ed ha associato un numero intero che rappresenta il livello di rischio della sua somministrazione e l'anno in cui è stato introdotto nel sistema sanitario. Ogni vaccino viene somministrato per prevenire una o più malattie, ed è di interesse conoscere tali malattie. Per ogni vaccino è di interesse conoscere la sua importanza. L'importanza di un vaccino è data da una funzione che calcola con la media del numero di persone del centro che hanno contratto le corrispondenti malattie.

Di ogni malattia interessa il codice identificativo ed il tipo (contagiosa, ereditaria, ecc.), mentre non è interessante risalire agli eventuali vaccini che la prevengono. Per una malattia può esistere un numero qualunque di vaccini che la prevengono.

Una vaccinazione rappresenta la somministrazione di un vaccino ad una persona (una persona viene sottoposta al massimo ad una somministrazione di ogni vaccino). Di ogni vaccinazione interessa la data in cui è avvenuta. Di ogni persona interessa il nome, il cognome, e la data di nascita, mentre non è di interesse risalire alla sue vaccinazioni. Al contrario, dato un vaccino è di interesse risalire alle vaccinazioni che hanno avuto per oggetto quel vaccino. Infine, di ogni malattia interessano le persone che le hanno contratte (mentre non è di interesse risalire dalle persone alle malattie che hanno contratto).

Le operazioni che si devono eseguire nella classe `CentroVaccinazioni` sono le normali operazioni di inserimento di oggetti (vaccini, persone, malattie, vaccinazioni), e di ispezione dei valori.

Esercizio 1 (punti 5) Si disegni il diagramma delle classi in UML per l'applicazione.

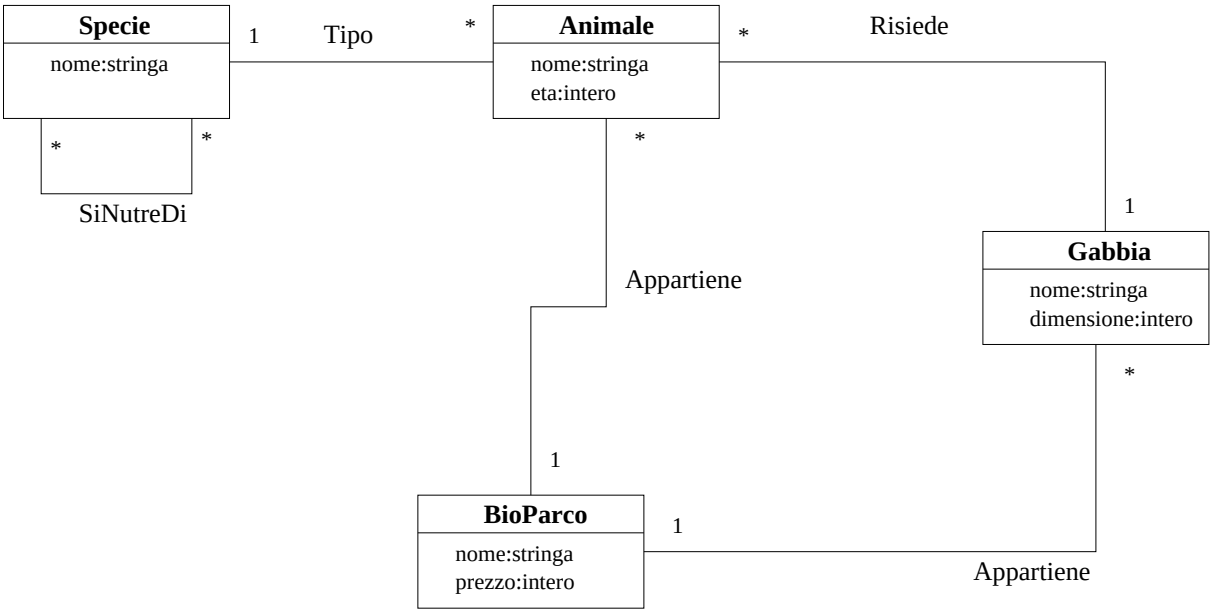
Esercizio 2 (punti 5) Si scrivano le definizioni delle classi C++ per l'applicazione. Si consideri disponibile la classe `Data` con tutte le funzioni di gestione che si ritengono necessarie.

Esercizio 3 (punti 15) Si scrivano le definizioni delle funzioni delle classi, gestendo le precondizioni attraverso la funzione `assert()`.

Esercizio 4 (punti 5) Si scriva una funzione esterna che prenda come parametri una data d, un numero intero n, ed un vettore v di vaccini, e restituisca il vettore contenente le persone alle quali è stata somministrata prima di d una vaccinazione di un vaccino presente in v la cui importanza è maggiore di n.

Compito del 2 dicembre 2003 (soluzione a pagina 132)

Si vuole progettare una classe C++ per la gestione di *Bioparchi* (giardini zoologici). Ciascun bioparco è caratterizzato da un insieme di animali ed un insieme di gabbie in cui gli animali risiedono. L'applicazione è descritta dal seguente schema in UML senza operazioni.



Si assuma che non possano esistere gabbie, animali o specie che abbiano lo stesso nome e che quindi la verifica di uguaglianza tra oggetti verta solo sul nome.

Le classi devono essere dotate solo delle normali operazioni di inserimento, ispezione e modifica dei dati. Tali operazioni possono essere realizzate nel modo che si ritiene più opportuno, ad eccezione delle operazioni di inserimento e modifica della classe BioParco che devono aderire alle seguenti specifiche:

InserisciAnimale($n : stringa, e : intero, s : Specie, g : Gabbia$) : *booleano* L'animale di nome n, età e e specie s viene inserito nel bioparco nella gabbia g. Se l'animale è già presente (anche in un'altra gabbia) allora la funzione non esegue modifiche e restituisce *false*; se invece l'inserimento va a buon fine la funzione restituisce *true*.
Precondizione: La gabbia g esiste nel bioparco.

SpostaAnimale($n : stringa, g : Gabbia$) L'animale di nome n, viene spostato nella gabbia g.
Precondizione: L'animale di nome n e la gabbia g esistono nel bioparco.

InserisciGabbia($g : Gabbia$) La gabbia g viene inserita nel bioparco.
Precondizione: La gabbia g non esiste nel bioparco.

Esercizio 1 (punti 10) Si scrivano le definizioni delle classi `BioParco`, `Animale` e `Specie` avendo già disponibile la classe `Gabbia` definita come segue.

```
class Gabbia
{
  friend ostream& operator<<(ostream& os, const Gabbia& g);
public:
  Gabbia(string n, unsigned d);
  string Nome() const { return nome; }
  unsigned Dimensione() const { return dimensione; }
private:
  string nome;
  unsigned dimensione;
};
```

Esercizio 2 (punti 10) Si scrivano le definizioni delle funzioni delle tre classi.

Esercizio 3 (punti 5) Si scriva una funzione esterna che prenda come parametro un bioparco e restituisca il numero totale di animali che sono in pericolo, cioè gli animali che hanno nella loro stessa gabbia uno o più animali di una specie che si nutre della loro specie.

Esercizio 4 (punti 5) Si aggiungano l'operatore di assegnazione e il costruttore di copia per la classe `BioParco` in modo che tutti gli oggetti collegati di tipo `Animale` (ma non quelli di tipo `Gabbia` e `Specie`) siano duplicati e non condivisi tra oggetti di tipo `BioParco`.

Compito del 9 gennaio 2004

Si vuole progettare una classe C++ per la gestione di un ufficio che eroga servizi al pubblico. Un ufficio si occupa di un insieme di servizi ed ha un insieme di sportelli (numerati da 0 a $n-1$) per la loro erogazione. Ciascuno sportello eroga un solo servizio, mentre un servizio può essere erogato da zero, uno o più sportelli. Ciascuno sportello serve un utente alla volta, mentre altri utenti possono essere in coda ad uno sportello. Ogni sportello ha una sua coda autonoma, gestita con la politica "primo arrivato, primo servito". Uno sportello può essere aperto, chiuso oppure in chiusura; quando uno sportello è chiuso non possono essere presenti utenti, mentre quando è in chiusura vengono serviti gli utenti in coda, ma non è permesso di aggiungerne di nuovi alla coda.

Di un servizio interessano il nome, la descrizione ed il tempo medio (in minuti, intero) per la sua erogazione. Dello sportello interessa il suo numero ed il nome dell'impiegato presente, oltre ovviamente al suo stato (aperto, chiuso o in chiusura). Le informazioni riguardanti gli utenti non sono note; si farà quindi riferimento alla classe `Utente` supposta già disponibile di cui non si conosce la definizione.

Le funzioni della classe che rappresenta l'ufficio sono specificate come segue:

ApriSportello(i : *intero*) Lo sportello i viene aperto.
Precondizione: Lo sportello i è chiuso.

ChiudiSportello(i : *intero*) Se non ci sono persone in coda lo sportello i viene chiuso, altrimenti viene posto in chiusura
Precondizione: Lo sportello i è aperto.

EsceUtente(i : *intero*) Il primo utente dello sportello i esce dall'ufficio, il primo in coda inizia ad essere servito. Se i è in chiusura e non ha utenti in coda allora i viene chiuso. Precondizione: Lo sportello i non è chiuso, ed ha almeno un utente presente.

EntraUtente(u : *utente*, s : *servizio*) : *intero* L'utente u viene inserito in fondo alla coda più corta (o una delle più corte) tra quelle degli sportelli aperti che erogano il servizio s. La funzione restituisce:

0 se l'operazione viene eseguita regolarmente
1 se non esistono sportelli che erogano il servizio s
2 se tutti gli sportelli che erogano il servizio s sono chiusi o in chiusura

Nel caso che la funzione restituisca 1 o 2, l'utente non viene inserito e l'operazione non ha effetto.

Esercizio 1 (punti 5) Si disegni il diagramma UML delle classi per l'applicazione descritta.

Esercizio 2 (punti 10) Si scrivano le definizioni delle classi C++ per l'applicazione, considerando disponibile la classe `Utente` di cui non conoscono però le funzioni.

Si noti che in base alle specifiche non è possibile inserire o eliminare sportelli o servizi in un ufficio. Tali informazione devono quindi essere tutte inserite per mezzo di un opportuno costruttore della classe.

Esercizio 3 (punti 10) Si scrivano le definizioni delle funzioni delle classi, gestendo le precondizioni attraverso la funzione `assert()`.

Esercizio 4 (punti 5) Si scriva una funzione esterna che prenda come parametro un ufficio e restituisca il tempo totale stimato per smaltire tutti gli utenti attualmente presenti. Si assuma che un utente normale impegni lo sportello per il tempo medio del servizio, mentre un utente esperto impegni lo sportello per un tempo pari al minimo tra 5 minuti e il tempo medio del servizio. A queste scopo si assuma che la classe `Utente` sia dotata della funzione `UtenteEsperto(Servizio* s)` che restituisce `true` se l'utente è esperto per il servizio puntato da `s`, restituisce `false` altrimenti.

Compito del 19 marzo 2004 **(soluzione a pagina 136)**

Si vuole progettare una classe C++ per realizzare una struttura dati a valori interi chiamata *Pila Bifronte*, in cui gli elementi possono essere inseriti ed eliminati solo dalle due estremità della struttura (chiamate testa e coda). Le operazioni principali della pila bifronte sono:

Push(el : *intero*, b : *booleano*) : Se b è *true* l'elemento el viene inserito in testa alla pila bifronte, altrimenti el viene inserito in coda.

Pop(b : *booleano*) : Se b è *true* l'elemento in testa alla pila viene eliminato, altrimenti viene eliminato l'elemento in coda
Precondizione: La pila bifronte non e vuota

Top(b : *booleano*) : *intero* : Se b è *true* restituisce l'elemento in testa alla pila, altrimenti restituisce l'elemento in coda.
Precondizione: La pila bifronte non e vuota

EstVuota(i : *intero*) : *booleano* : Restituisce *true* se la pila bifronte è vuota, falso altrimenti.

La progettazione deve inoltre attenersi alle seguenti scelte progettuali:

- Al momento della costruzione la pila bifronte è vuota, ma il numero di elementi nella pila può crescere in modo indefinito in entrambe le direzioni.
- Non si possono utilizzare le classi della libreria standard STL (ad es. `vector`), ma si devono utilizzare unicamente vettori dinamici allocati esplicitamente utilizzando l'operatore `new`.
- Bisogna prevedere anche la possibilità di copia tra oggetti, quindi è indispensabile definire le funzioni speciali (costruttore di copia, operatore di assegnazione e distruttore) che evitino la condivisione di memoria tra oggetti della classe.

Esercizio 1 (punti 8) Si scriva la definizione della classe `PilaBifronte`.

Esercizio 2 (punti 18) Si scrivano le definizioni delle funzioni della classe `PilaBifronte`.

Esercizio 3 (punti 4) Si scriva una funzione esterna alla classe `PilaBifronte` che prenda come parametri per riferimento due oggetti della classe `pb1` e `pb2` e modifichi `pb1` inserendo in testa il più piccolo tra l'elemento in testa e quello in coda di `pb2`. Se tale elemento non esiste, la funzione deve inserire in coda a `pb1` il valore `0`.

Compito del 28 giugno 2004 (soluzione a pagina 141)

Si vuole progettare la classe C++ `VettoreCompatto` per memorizzare vettori di interi in modo compatto, sfruttando le ripetizioni di caratteri consecutivi. Ad esempio, il vettore $[4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 12, 12, 4, 4]$ viene memorizzato attraverso la sequenza di coppie $\langle(4, 7), (5, 6), (12, 2), (4, 2)\rangle$, dove il primo elemento della coppia è il valore e il secondo il numero di ripetizioni consecutive.

Le operazioni principali da realizzare nella classe sono:

Inserisci(el : *intero*) : L'elemento *el* viene aggiunto come ultimo elemento del vettore.

Elimina() : L'ultimo elemento del vettore viene eliminato.
Precondizione: Il vettore non è vuoto

Elem(i : *intero*) : *intero* : Restituisce l'elemento i-esimo del vettore.
Precondizione: Il vettore non è vuoto ed i è compreso tra 0 e il numero di elementi del vettore meno uno.

NumElem() : *intero* : Restituisce il numero di elementi del vettore.

La progettazione deve inoltre attenersi alle seguenti scelte progettuali:

- Al momento della costruzione il vettore è vuoto, ma il suo numero di elementi può crescere in modo indefinito.
- Non si possono utilizzare le classi della libreria standard STL (ad es. `vector`), ma si devono utilizzare unicamente vettori dinamici allocati esplicitamente utilizzando l'operatore `new`.
- Bisogna prevedere anche la possibilità di copia tra oggetti, quindi è indispensabile definire le funzioni speciali (costruttore di copia, operatore di assegnazione e distruttore) che evitino la condivisione di memoria tra oggetti della classe.

- L'operazione *Elem* deve essere implementata ridefinendo l'operatore `[]` per la classe `VettoreCompatto`.

Esercizio 1 (punti 8) Si scriva la definizione della classe `VettoreCompatto`.

Esercizio 2 (punti 16) Si scrivano le definizioni delle funzioni della classe `VettoreCompatto`.

Esercizio 3 (punti 6) Si scriva una funzione esterna alla classe `VettoreCompatto` che prenda come parametro un oggetto della classe e restituisca un nuovo oggetto ottenuto invertendo l'ordine degli elementi del vettore dell'oggetto passato come parametro.

Compito del 8 luglio 2004

L'applicazione da progettare riguarda le informazioni su facoltà, docenti e corsi di una certa università. Di ogni docente interessa il nome ed il cognome, la facoltà di cui è dipendente e da che anno, i corsi che insegna e da che anno. Di ogni corso interessa il nome. Di ogni facoltà interessa il nome e il preside, che è un docente della stessa università, ma non necessariamente della stessa facoltà. Un docente può essere preside di più facoltà. Ogni facoltà ha almeno otto docenti fra i suoi dipendenti. Ogni docente insegna fra uno e tre corsi, mentre ogni corso è insegnato da almeno un docente.

Ogni docente può essere in servizio oppure assente, in particolare per ferie, malattia o anno sabbatico. Dall'anno sabbatico si può tornare solamente in servizio, mentre se se si è in ferie si può anche andare in malattia e se si è in malattia si può anche andare in anno sabbatico.

Esercizio 1 (punti 5) Si disegni il diagramma UML delle classi per l'applicazione descritta.

Esercizio 2 (punti 3) Si disegni il diagramma UML degli stati per lo stato di servizio di un docente.

Esercizio 3 (punti 8) Si scrivano le definizioni delle classi C++ per l'applicazione, considerando solo le normali operazioni di inserimento, cancellazione, modifica e ispezione dei dati (scegliere quelle che si ritengono opportune).

Esercizio 4 (punti 10) Si scrivano le definizioni delle funzioni delle classi (anche solo quelle che si ritengono più significative).

Esercizio 5 (punti 4) Si scriva una funzione esterna che prenda come parametro un vettore di facoltà, e calcoli la percentuale di quelle il cui preside non è loro dipendente.

Compito del 14 settembre 2004

Si vuole progettare una classe C++ per la gestione dei lavori di una *commissione*. La commissione ha dei componenti, e di ciascun componente interessa il nome e la data in cui è entrato a far parte della commissione. La commissione svolge delle riunioni. Ciascuna riunione è identificata da una data ed un argomento discusso.

Le operazioni da realizzare per la classe sono le normali operazioni di modifica e ispezione dei dati, con le seguenti restrizioni:

- È possibile inserire un componente o sostituirlo con un altro componente, ma non si possono eliminare componenti.
- Le riunioni vengono inserite nell'ordine in cui si svolgono e non possono essere modificate o eliminate. All'atto dell'inserimento di una riunione è necessario controllare la precondizione che la data sia posteriore a quella delle riunioni precedenti.
- Tra una riunione e l'altra è possibile inserire o sostituire al massimo un componente. La classe deve rifiutare modifiche che non rispettino questa precondizione.
- Per poter svolgere una riunione, la commissione deve avere almeno due componenti.

- All'atto della creazione di una commissione viene specificato un solo componente.

Si consideri disponibile già scritta la classe `Data` per la gestione delle date, dotata di tutte le funzioni e gli operatori che si ritengono opportuni.

Esercizio 1 (punti 8) Si scriva la definizione della classe `Commissione`.

Esercizio 2 (punti 16) Si scrivano le definizioni delle funzioni della classe `Commissione`.

Esercizio 3 (punti 6) Si scriva una funzione esterna alla classe `Commissione` che prenda come parametri un oggetto della classe ed un argomento e calcoli la distanza massima in giorni tra due riunioni consecutive su quell'argomento.

Compito del 6 dicembre 2004 (soluzione a pagina 145)

Si consideri il linguaggio di programmazione YATUL, i cui programmi sono composti da un insieme di *dichiarazioni* ed una sequenza di *istruzioni*.

Una dichiarazione YATUL è composta da un nome di variabile (stringa alfabetica) ed il suo valore iniziale (numero intero). Un'istruzione è composta da due operandi (stringhe alfabetiche), un operatore (carattere) ed un risultato (stringa alfabetica).

Gli operatori ammessi da YATUL sono solo il `'+'` e il `'-'`. Gli operatori hanno l'ovvio significato di somma e sottrazione, e l'istruzione ha il significato dell'assegnazione alla variabile risultato del valore dell'operazione sui due operandi.

Come esempio si consideri il seguente programma YATUL (qui mostrato nel formato con cui viene scritto su un file, la prima riga contiene il numero di dichiarazioni ed il numero di istruzioni).

```
#yatul 5/4
alfa = 3
beta = 9
gamma = 0
delta = 6
theta = -4
alfa <-- beta + alfa
gamma <-- beta - alfa
delta <-- gamma + theta
alfa <-- gamma - alfa
```

Si vuole progettare la classe C++ `Yatul`, in modo tale che ogni oggetto della classe corrisponda ad un programma YATUL. Le operazioni principali della classe sono le seguenti.

InserisciDichiarazione(*d* : *Dichiarazione*)
: La dichiarazione *d* viene aggiunta al programma.
Precondizione: Non esiste già nel programma una dichiarazione che riguarda la variabile dichiarata da *d*

InserisciIstruzione(*i* : *Istruzione*)
: L'istruzione *i* viene inserita come ultima istruzione del programma.

ProgrammaCorretto() : *booleano*
: Restituisce *true* se tutte le variabili presenti nelle istruzioni del programma (come operando o come risultato) sono presenti nelle dichiarazioni del programma, *false* altrimenti.

CalcolaValore(*var* : *stringa*) : *intero*
: Calcola il valore finale che l'esecuzione del programma assegna alla variabile *var*. Per esecuzione del programma si intende l'esecuzione sequenziale di tutte le istruzioni (facendo evolvere i valori delle variabili a partire dai valori contenuti nelle dichiarazioni).
Precondizione: *var* è presente in una dichiarazione del programma ed il programma è corretto.

Per esempio, nel programma precedente il valore calcolato per la variabile `alfa` è `-15`, mentre quello per `beta` è `9`.

Esercizio 1 (punti 6) Assumendo disponibili, già realizzate, le classi `Dichiarazione` e `Istruzione` definite in seguito, si scriva la definizione della classe `Yatul`. Si doti la classe `Yatul` di tutte le funzioni di ispezione dei dati che si ritengono opportune (vedere anche esercizio 5).

Le definizioni delle classi `Dichiarazione` e `Istruzione` sono le seguenti.

```
class Dichiarazione
{
 public:
  Dichiarazione() {}
  Dichiarazione(string n, int v)
    : nome(n) { valore = v; }
  string Nome() const { return nome; }
  int Valore() const { return valore; }
  void SetValore(int v) { valore = v; }
 private:
  string nome;
  int valore;
};
```

```
class Istruzione
{
  public:
   Istruzione() {}
   Istruzione(string o1, string o2, char o, string r)
     : operando1(o1), operando2(o2), risultato(r)
     { operatore = o; }
   char Operatore() const { return operatore; }
   string Operando1() const { return operando1; }
   string Operando2() const { return operando2; }
   string Risultato() const { return risultato; }
  private:
   char operatore;
   string operando1, operando2, risultato;
};
```

Esercizio 2 (punti 12) Si scrivano le definizioni delle funzioni della classe `Yatul`, incluso un costruttore senza argomenti che costruisce un programma senza istruzioni e con una sola dichiarazione che dichiara la variabile `alfa` con il valore `0`.

Esercizio 3 (punti 3) Si scrivano gli operatori di output (*friend*) per le tre classi `Dichiarazione`, `Istruzione` e `Yatul` che scrivano i dati nel formato mostrato dall'esempio.

Esercizio 4 (punti 5) Si scrivano gli operatori di input (*friend*) per le tre classi `Dichiarazione`, `Istruzione` e `Yatul` che leggano i dati nel formato mostrato dall'esempio.

Esercizio 5 (punti 4) Si scriva una funzione esterna che prenda come parametro un oggetto della classe `Yatul` e restituisca *true* se nel programma esiste almeno un variabile *inutile*, $false$ altrimenti. Una variabile è inutile se non compare mai come operando di una istruzione. Per il programma dell'esempio la funzione deve restituire *true* in quando la variabile `delta` è inutile.

Compito del 20 dicembre 2004 **(soluzione a pagina 150)**

Si vuole progettare una classe per la gestione di una partita del gioco (solitario) *YAIG*. In una partita di YAIG, il giocatore deve muoversi in una scacchiera quadrata di dimensione arbitraria. Sulla scacchiera sono posizionate in modo casuale alcune *bombe* ed un *tesoro*. Scopo del giocatore è di raggiungere la posizione del tesoro senza *scoppiare* (cioè muoversi su una casella occupata da una bomba) per più di un numero fissato di volte. Il giocatore parte dalla casella (0,0) e si muove di una casella alla volta in direzione orizzontale o verticale. La direzione è identificata da un carattere che rappresenta un punto cardinale: 'N' (Nord) colonna crescente, 'E' (Est) riga crescente,'S' (Sud) colonna decrescente, 'O' (Ovest) riga decrescente. Le operazioni principali della classe sono le seguenti.

$Muovi(d:\ carattere):\ intero$

Il giocatore si muove di una casella nella direzione identificata da d. In base a cosa è presente nella casella di arrivo la funzione ha uno dei seguenti effetti:

- La casella è vuota: il giocatore si porta in essa e la funzione restituisce 0.
- E' presente una bomba: il giocatore rimane nella sua posizione, il contatore degli scoppi viene incrementato e se raggiunge il massimo consentito la partita finisce e la funzione restituisce 2, altrimenti la funzione restituisce 1.
- E' presente il tesoro, la partita finisce con la vittoria e la funzione restituisce 3.

Se la mossa porta il giocatore fuori della scacchiera allora la mossa non ha effetto e la funzione restituisce -1.
Precondizione: $d \in \{'N','S','E','O'\}$

Esplora(r : *intero*) : *intero*

Restituisce il numero di bombe presenti in un area della scacchiera di raggio r intorno alla posizione del giocatore.

All'inizio del gioco vengono scelte la dimensione della scacchiera e il numero di bombe, e si genera una configurazione iniziale casuale. Il numero massimo di scoppi viene fissato automaticamente pari alla metà del numero di bombe. Il numero di bombe non può superare la metà delle caselle della scacchiera.

Esercizio 1 (punti 6) Assumendo disponibile, già realizzata, la classe `Casella` definita in seguito, si scriva la definizione della classe `Yaig`. Si includa un costruttore che, prendendo i parametri opportuni, crea una configurazione iniziale. Si doti la classe `Yaig` di tutte le funzioni di ispezione dei dati che si ritengono opportune. La definizione della classe `Casella` è la seguente.

```
class Casella
{
  friend bool operator==(const Casella& c1, const Casella& c2);
  friend ostream& operator<<(ostream& os, const Casella& c);
 public:
  Casella(int i = 0, int j = 0) { riga = i; colonna = j; }
  int Riga() const { return riga; }
  int Colonna() const { return colonna; }
  void Set(int i, int j) { riga = i; colonna = j; }
 private:
  int riga, colonna;
};
```

Esercizio 2 (punti 16) Si scrivano le definizioni delle funzioni della classe `Yaig`. Si consideri disponibile la funzione `Random(int a,int b)` che restituisce un valore intero casuale tra `a` e `b` compresi (con la precondizione `a <= b`).

Esercizio 3 (punti 8) Si scriva un programma che esegua una partita del gioco utilizzando la classe `Yaig`. Il programma deve far scegliere i parametri della partita all'utente (da tastiera), ed eseguire un ciclo che termina con la fine della partita in cui ad ogni iterazione il giocatore sceglie se effettuare una mossa oppure una esplorazione. Il programma deve anche mostra all'utente ad ogni iterazione la sua posizione corrente, il numero di mosse effettuate ed il numero di scoppi rimasti prima della fine.

Compito del 15 luglio 2005

Si vuole progettare una classe C++ per la gestione dei lavori del *comitato scientifico* di un congresso per la selezione degli articoli proposti da presentare al congresso.

Il comitato ha dei membri, di cui interessa il nome e l'affiliazione, e degli articoli proposti per la pubblicazione, di cui interessa il titolo e il nome del primo autore.

A ciascun membro del comitato vengono assegnati da 5 a 10 articoli proposti di cui deve fare la valutazione. Ciascun articolo deve essere valutato da almeno tre membri del comitato. Ciascuna valutazione comprende un voto (da 0 a 10) ed un commento da mandare agli autori.

Le operazioni principali da realizzare per la classe sono le seguenti:

- Inserire un articolo o un membro del comitato, con nessuna assegnazione iniziale.
- Assegnazione di un articolo ad un membro (precondizione: entrambi esistono e non sono assegnati l'uno all'altro)
- Inserimento di una valutazione (precondizione: l'articolo è assegnato al membro).
- Verifica che un articolo sia stato assegnato ad almeno tre membri (precondizione: l'articolo esiste).
- Verifica che un articolo sia stato giudicato da almeno tre membri (precondizione: l'articolo esiste). Se la verifica è positiva, la funzione deve anche restituire la media dei voti delle valutazioni.

Esercizio 1 (punti 6) Si disegni il diagramma delle classi in UML per l'applicazione.

Esercizio 2 (punti 6) Si scriva le definizioni delle classi coinvolte.

Esercizio 3 (punti 12) Si scrivano le definizioni di *alcune* delle funzioni delle classi coinvolte.

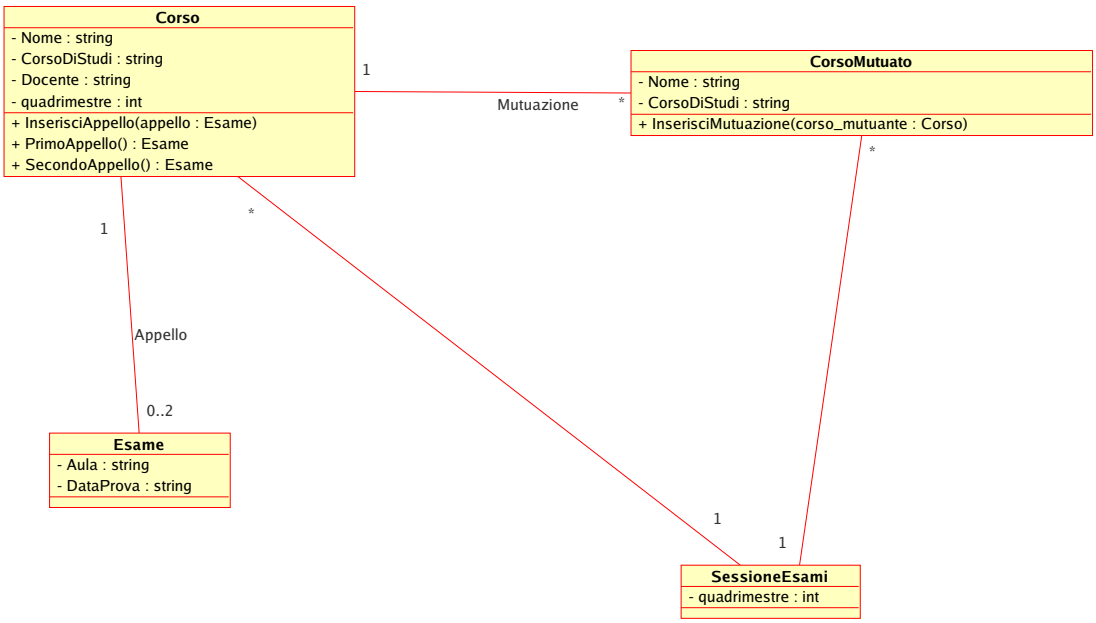
Esercizio 4 (punti 6) Si scriva una funzione esterna che prenda come parametro un comitato scientifico e restituisca:

- il numero di articoli che non sono ancora stati assegnati a tre membri;
- il numero di articoli con non sono ancora stati valutati da tre membri
- il numero di articoli che sono stati valutati da tre membri, ma che almeno una delle valutazioni ha un commento più breve di 100 caratteri.

Compito del 5 dicembre 2005 (soluzione a pagina 153)

L'applicazione da progettare riguarda la gestione di una sessione di esami di una facoltà universitaria. Una sessione comprende i corsi (identificati da nome e corso di studi) e i relativi esami, oltre ai corsi mutuati, che sono corsi che non vengono erogati direttamente ma confluiscono in un altro corso. I corsi mutuati non hanno appelli di esame in quanto i loro esami sono implicitamente assegnati nella stessa data del corso mutuante.

Lo schema UML corrispondente è il seguente.



Le operazioni della classe SessioneEsami sono le seguenti:

$InserisciCorso(n : stringa, cs : stringa, d : stringa, q : intero)$

Il corso di nome n, corso di studi cs, docente d e quadrimestre q viene inserito nella sessione d'esami.
Precondizione: non esiste già un corso (mutuato o meno) con lo stesso nome e lo stesso corso di studi.

$InserisciCorsoMutuato(n : stringa, cs : stringa, nm : stringa, csm : stringa)$

Il corso di nome n e corso di studi cs viene inserito come corso mutuato dal corso di nome n e corso di studi csm.
Precondizione: non esiste già un corso (mutuato o meno) con lo stesso nome e lo stesso corso di studi, esiste il corso di nome n e corso di studi csm (non mutuato).

InserisciAppelloCorso(n : *stringa*, cs : *stringa*, d : *Data*, a : *stringa*)
Viene inserito un appello in data d a aula a per il corso di nome n e corso di studi cs.
Precondizione: Il corso di nome n e corso di studi cs ha meno di 2 appelli. Se il corso ha già un appello, il nuovo appello deve essere in data diversa.

EliminaCorso(n : *stringa*, cs : *stringa*)
Il corso di nome n e corso di studi cs viene eliminato dalla sessione. Anche tutti gli esami del corso e tutti i corsi mutuati da questo vengono eliminati.
Precondizione: Esiste un corso di nome n e corso di studi cs.

EliminaCorsoMutuato(n : *stringa*, cs : *stringa*)
Il corso mutuato di nome n e corso di studi cs viene eliminato dalla sessione.
Precondizione: Esiste un corso mutuato di nome n e corso di studi cs.

AuleOccupate(d : *data*) : *vector*$\langle$*stringa*$\rangle$
Restituisce il vettore delle aule occupate da esami nella data d.

Si noti che la classe `Corso` ha due selettore specifici per il primo (con data precedente) e il secondo appello (data successiva). Questi selettori hanno la precondizione che siano stati definiti un numero sufficiente di esami.

Esercizio 1 (punti 6) Assumendo disponibili, già realizzate, la classe `Esame` definita in seguito e la classe `Data` (con le funzioni e gli operatori che si ritengono opportuni) si scriva la definizione delle classi `Corso`, `CorsoMutuato` e `SessioneEsami`. Si dotino le classi definite delle funzioni di ispezione dei dati che si ritengono opportune (vedere anche esercizio 3).

La definizione della classe `Esame` è la seguente.

```
class Esame
{public:
  Esame() {}
  Esame(Data d, string a) : data(d), aula(a) {}
  string Aula() const { return aula; }
  Data DataProva() const { return data; }
 private:
  Data data;
  string aula;
};
```

Esercizio 2 (punti 10) Si scrivano le definizioni delle funzioni delle classi `Corso`, `CorsoMutuato` e `SessioneEsami`.

Esercizio 3 (punti 5) Si scriva la funzione esterna `VerificaCorrettezza()` che prenda come parametro un oggetto della classe `SessioneEsame` e restituisca *true* se e solo se sono vere entrambe le seguenti condizioni:

- i corsi erogati nello stesso quadrimestre della sessione di esami hanno esattamente due appelli;
- i corsi erogati in un quadrimestre diverso da quello della sessione di esami hanno esattamente un appello.

Esercizio 4 (punti 5) Siccome la classe `SessioneEsami` dovrà gestire la creazione di oggetti dinamici al suo interno, sarà necessaria per questa classe la definizione delle funzioni speciali: costruttore di copia, operatore di assegnazione e distruttore. Si scrivano le definizioni del distruttore che deallochi tutti gli oggetti collegati all'oggetto di tipo `SessioneEsami` e del costruttore di copia che faccia la copia profonda di tutti gli oggetti collegati. La definizione dell'operatore di assegnazione non è richiesta.

Esercizio 5 (punti 4) Si scriva la definizione (*friend*) dell'operatore di input per la classe `SessioneEsami` che legga i dati nel formato mostrato dall'esempio seguente. L'operatore di output non è richiesto.

```
Quadrimestre: 2
Corsi: 3
MatematicaI Ambiente Rossi 2
```

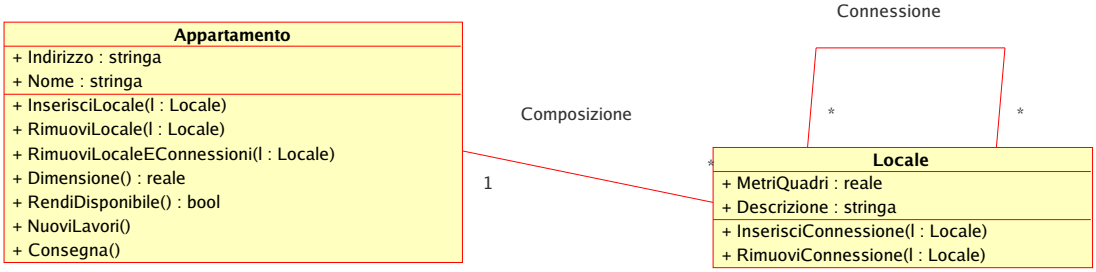
```
MatematicaII Ambiente Rossi 2
Informatica Civile Bianchi 1
CorsiMutuati: 2
MatematicaI Civile MatematicaI Ambiente
MatematicaII Civile MatematicaII Ambiente
Esami: 5
MatematicaI Ambiente 12/9/2005 A
MatematicaI Ambiente 20/9/2005 F
MatematicaII Ambiente 14/9/2005 51
MatematicaII Ambiente 18/9/2005 38
Informatica Civile 12/9/2005 B
```

Compito del 21 dicembre 2005 (soluzione a pagina 164)

L'applicazione di interesse riguarda la progettazione di appartamenti. Di ogni appartamento interessano l'indirizzo, il nome, i locali che lo compongono e come questi sono connessi tra loro (si noti che se il locale A e connesso con il locale B, allora anche B e connesso con A). Di ciascun locale interessano il codice identificativo (di 3 caratteri), i metri quadri ed una descrizione testuale, tipo `salotto`, `bagno`, `cucina`, ...

Un appartamento è inizialmente in preparazione, poi alla fine dei lavori diviene pronto per la consegna; a questo punto possono essere richiesti ulteriori lavori (e quindi torna ad essere in preparazione) o può essere consegnato. Solo quando un appartamento è in preparazione si possono aggiungere ed eliminare locali da esso.

Lo schema UML corrispondente è il seguente.



Le operazioni della classe Appartamento sono specificate come segue:

InserisciLocale(l : *Locale*)
: Il locale l viene inserito nell'appartamento.
Precondizione: Il locale l non è presente nell'appartamento (cioè non è presente un locale con lo stesso codice). L'appartamento è in lavorazione.

RimuoviLocale(l : *Locale*)
: Il locale l viene eliminato nell'appartamento.
Precondizione: Il locale l è presente nell'appartamento. L'appartamento è in lavorazione.

RimuoviLocaleEConnessioni(l : *Locale*)
: Il locale l viene eliminato nell'appartamento e vengono rimosse tutte le sue connessioni (ma non i locali connessi) a locali presenti nell'appartamento.
Precondizione: Il locale l è presente nell'appartamento. L'appartamento è in lavorazione.

Dimensione() : *reale*
: Restituisce la dimensione totale dell'appartamento ottenuta come somma dei metri quadri dei suoi locali.

RendiPronto() : *booleano*
: Se i locali dell'appartamento sono connessi solo ad altri locali dell'appartamento e non ad altri, l'appartamento passa nello stato di pronto e viene restituito *true*; altrimenti lo stato non cambia e viene restituito *false*.
Precondizione: L'appartamento è in lavorazione.

NuoviLavori()
L'appartamento passa in lavorazione.
Precondizione: L'appartamento è pronto.

Consegna()
L'appartamento viene consegnato.
Precondizione: L'appartamento è pronto.

Esercizio 1 (punti 2) Si tracci il diagramma degli stati per la classe `Appartamento`.

Esercizio 2 (punti 6) Si scriva la definizione delle classi `Locale` e `Appartamento`. Si dotino le classi definite delle funzioni di ispezione dei dati che si ritengono opportune.

Esercizio 3 (punti 10) Si scrivano le definizioni delle funzioni delle classi `Locale` e `Appartamento`.

Esercizio 4 (punti 4) Si scriva la funzione esterna `VerificaAbitabilita()` che prenda come parametro un oggetto della classe `Appartamento` e restituisca *true* se e solo se nell'appartamento è presente esattamente una cucina e almeno un bagno.

Esercizio 5 (punti 4) Si scriva la definizione (*friend*) dell'operatore di output per la classe `Appartamento` che scriva tutti i dati dell'appartamento comprese le connessioni dei suoi locali.

Compito del del 29 marzo 2006

L'applicazione da progettare riguarda le informazioni sulle visite mediche effettuate in una certa ASL. Di ogni visita interessa l'assistito a cui e stata effettuata, la data di effettuazione e le eventuali prescrizioni indicate. Di ogni assistito interessa il codice, il nome, il cognome e la data di nascita. Di ogni prescrizione interessa il codice e la visita in cui è stata indicata.

Le visite possono essere soltanto inserite nella ASL, ma mai eliminate. Una visita inserita è inizialmente *prenotata*; successivamente può essere *disdetta* o *effettuata*. Le prescrizioni possono essere aggiunte o eliminate alle visite prenotate, non alle visite disdette o effettuate. Una visita non può essere disdetta se ci sono delle prescrizioni, ma può essere effettuata.

Esercizio 1 (punti 5) Si disegni il diagramma UML delle classi per l'applicazione descritta e la specifica delle operazioni principali.

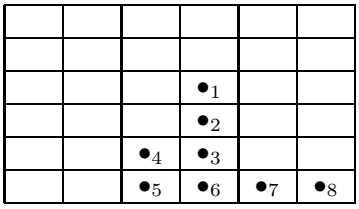
Esercizio 2 (punti 10) Si scrivano le definizioni delle classi C++ per l'applicazione. Si consideri la classe `Data` già disponibile, con le funzioni e gli operatori che si ritengono opportuni.

Esercizio 3 (punti 10) Si scrivano le definizioni di alcune delle funzioni delle classi (quelle che si ritengono più significative), gestendo le precondizioni attraverso la funzione `assert()`.

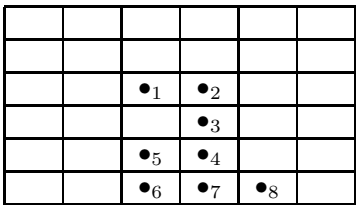
Esercizio 4 (punti 5) Si scriva una funzione esterna che prenda come parametri una ASL, una persona e un anno e restituisca il numero totale di farmaci che sono stati prescritti alla persona durante l'anno.

Compito del del 10 luglio 2006

L'applicazione da progettare riguarda il gioco *Snake*. Il gioco è composto da una scacchiera quadrata $n \times n$ e da un *serpente* che occupa una sequenza di k caselle. Il serpente è posizionato sulla scacchiera in modo tale che la prima casella, detta *testa*, è adiacente (cioè ha un lato in comune) alla seconda, e ciascuna casella del corpo è adiacente alla successiva. Come esempio si consideri la seguente scacchiera 6×6 con un serpente di lunghezza 8 in cui il simbolo $\bullet_i$ rappresenta l'i-esimo segmento del serpente ($\bullet_1$ è la testa).



Il serpente si può muovere spostando la testa in una delle caselle ad essa adiacenti seguendo le 4 direzioni (`N`, `E`, `S` e `O`). Le caselle del corpo vanno a posizionarsi nelle casella precedentemente occupata dal pezzo di corpo precedente. Ad esempio, una mossa in direzione `O` porta il serpente nella posizione seguente.



Una mossa che porti la testa del serpente fuori dalla scacchiera, oppure sopra un altro pezzo del serpente stesso, porta alla *morte* del serpente ed alla conseguente fine della partita. Mosse eseguite quando il serpente è morto non sono ammesse.

Esercizio 1 (punti 10) Si scriva la definizione della classe `Snake` che permette di gestire una partita del gioco eseguendo le mosse che vengono invocate attraverso un'apposita funzione della classe. La classe deve inoltre essere dotata di un unico costruttore con due argomenti interi n e k (precondizione: $k \leq 2n$) che costruisce una scacchiera di dimensione $n \times n$ con un serpente di dimensione k posizionato in un modo scelto dallo studente. La classe deve inoltre mettere a disposizione una funzione che *resusciti* il serpente nella posizione in cui si trovava prima della mossa che ne ha causato la morte.

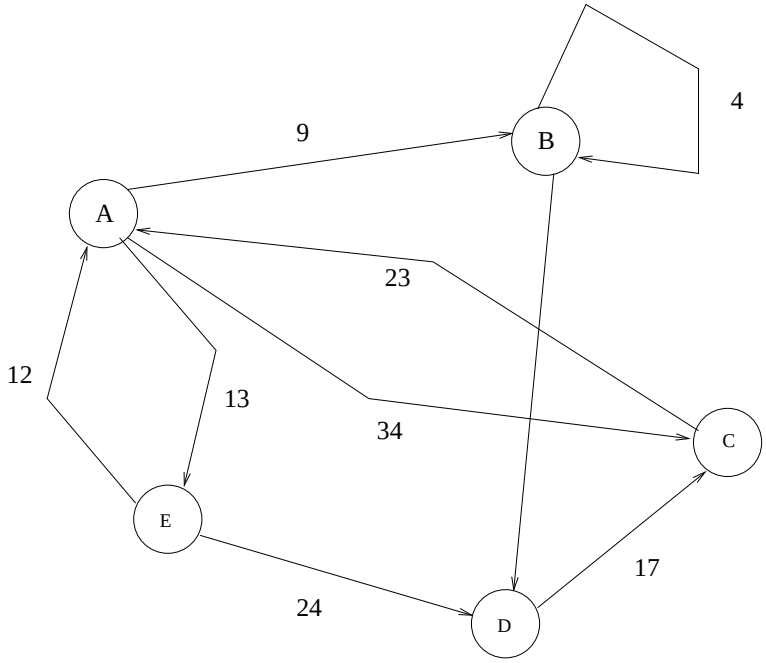
Esercizio 2 (punti 10) Si scriva la definizione delle funzioni delle classe `Snake` dotandola dei selettori che si ritengono opportuni per l'ispezione completa dello stato della scacchiera. Si gestiscano le precondizioni attraverso la funzione `assert()`.

Esercizio 3 (punti 5) Si scriva una funzione esterna alla classe che prenda come parametri un oggetto di classe `Snake` ed una stringa composta dai soli caratteri {`'N'`,`'E'`,`'S'`,`'O'`} ed esegua la sequenza di mosse rappresentata dalla stringa (o la parte di essa eseguibile fino alla morte del serpente).

Esercizio 4 (punti 5) Si scriva la definizione (*friend*) degli operatori di input e output per la classe `Snake` che leggano e scrivano i dati nel formato che si ritiene opportuno (lo stesso per entrambi).

Compito del 6 settembre 2006

Un *grafo orientato* è un insieme di nodi ed un insieme di archi, che sono coppie ordinate di nodi. I nodi sono identificati da una etichetta (stringa), mentre ogni arco ha associato un peso (intero positivo). Come esempio, si consideri il seguente grafo.



Si richiede di progettare una classe per la gestione dei grafi orientati. Le operazioni possibili sui grafi sono l'inserimento e la cancellazione di archi e l'inserimento e la cancellazione di nodi. Un nodo può essere eliminato solo se è *isolato*, cioè non ha archi né entranti né uscenti. Un arco ovviamente può essere inserito solo se esistono nel grafo i due nodi corrispondenti e l'arco non esiste già. Il nodo iniziale e quello finale di un arco possono anche coincidere.

Esercizio 1 (punti 10) Si scriva la definizione della classe `Grafo` che permette di gestire un grafo. La classe deve essere dotata di un unico costruttore senza argomenti che crea il grafo vuoto (senza nodi e senza archi). La classe deve ovviamente includere le funzioni per l'ispezione completa del grafo.

Esercizio 2 (punti 10) Si scriva la definizione delle funzioni delle classe `Grafo`. Si gestiscano le precondizioni attraverso la funzione `assert()`.

Esercizio 3 (punti 5) Si scriva l'operatore `==` di uguaglianza (*friend*) che verifica se due oggetti della classe `Grafo` sono identici (stessi nodi, stessi archi e stessi pesi).

Esercizio 4 (punti 5) Si scriva una funzione esterna alla classe `Grafo` che prenda un grafo e due etichette di nodi e verifichi se esiste un cammino orientato di lunghezza 3 (cioè una sequenza di tre archi) che congiunge il primo nodo al secondo.

Compito del 11 dicembre 2006 (soluzione a pagina 169)

Si vuole progettare un'applicazione per la gestione delle Scuole dell'Infanzia (per i bambini dai 3 ai 5 anni). Ciascuna scuola ha un nome ed un circolo scolastico di riferimento. La scuola è composta da un insieme di classi. Ciascuna classe della scuola è identificata da una lettera (A, B, ...) ed è frequentata da un certe numero di allievi (massimo 28). Di ciascun allievo interessa il nome (che lo identifica), il livello (bambino piccolo, bambino medio, bambino grande) e il numero di telefono di un genitore. Un bambino frequenta una sola classe. In ciascuna classe insegnano dei maestri, dei quali interessa in nome, il sesso, la data di nascita e il titolo di studio. Un maestro può insegnare anche in più classi e in scuole diverse.

Le operazioni fondamentali della classe Scuola sono le seguenti.

InserisciAllievo(c : *carattere*, a : *Allievo*)

L'allievo a viene iscritto nella scuola nella classe identificata dalla lettera c.

Precondizioni: L'allievo a non è già iscritto alla scuola. La classe c esiste e non è piena.

EliminaAllievo(a : *Allievo*)

L'allievo a viene cancellato dalla scuola. Se l'allievo non è presente in alcuna classe della scuola, l'operazione non ha alcun effetto.

AssegnaMaestro($c : carattere, m : Maestro$)
Il maestro m viene assegnato alla classe identificata dalla lettera c.
Precondizioni: la classe c esiste. Il maestro m non insegnava già nella classe c.

RevocaMaestro($m : Maestro$)
Il maestro m viene eliminato da ciascuna classe a cui insegnava.

CreaClasse()
Viene creata una nuova classe a cui viene data la lettera identificativa successiva all'ultima presente nella scuola. Se la scuola aveva già almeno due allievi, un allievo (scelto arbitrariamente) viene trasferito dalla classe con il massimo numero di bambini alla nuova classe. Il bambino trasferito deve essere del livello di maggior presenza nella classe di partenza.

TotaleAllievi() **:** intero
Restituisce il numero totale di allievi frequentanti la scuola.

Esercizio 1 (punti 7) Si tracci il diagramma UML delle classi identificando anche le operazioni fondamentali delle altre classi.

Esercizio 2 (punti 7) Si scriva la definizione delle classi C++ che compongono l'applicazione.

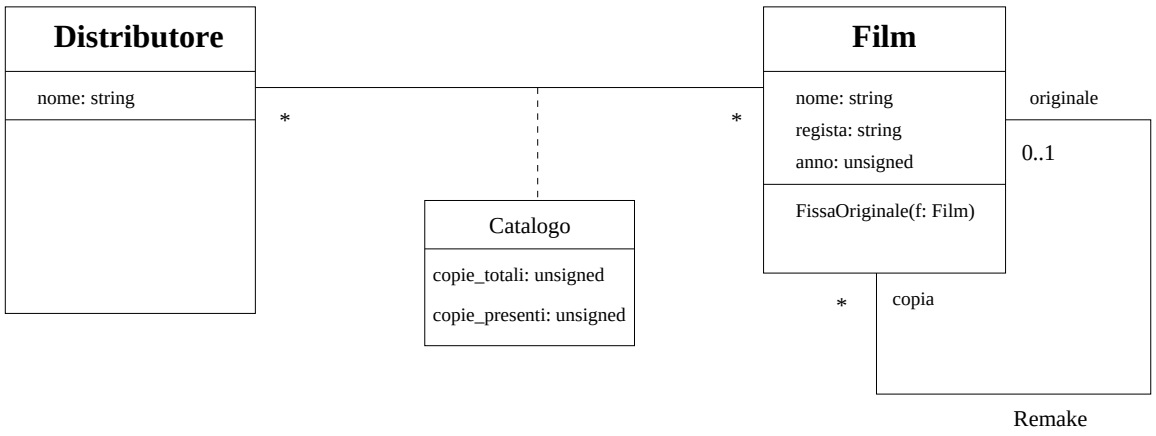
Esercizio 3 (punti 12) Si Scriva le definizione delle funzioni membro delle classi che compongono l'applicazione.

Esercizio 4 (punti 4) Si scriva una funzione esterna che prende come parametro un oggetto di tipo `Scuola` e restituisce un vettore di booleani di dimensione pari al numero di classi della scuola. La funzione calcola quali classi della scuola sono omogenee (cioè contengono tutti bambini dello stesso livello).

Compito del 10 gennaio 2007

Si vuole progettare un'applicazione per la gestione di distributori automatici per il noleggio di film in DVD. Di un film interessano il titolo, il regista e l'anno di uscita. Inoltre, un film è identificato dalla coppia titolo-regista. Per la nostra applicazione, è anche importante memorizzare se un film sia eventualmente il *remake* di un altro film. Un distributore ha un certo numero di copie di ciascun film disponibili per il noleggio.

Lo schema UML dell'applicazione è il seguente (dove le operazioni della classe Distributore sono state omesse)



L'operazione FissaOriginale della classe Film assegna ad un film (che quindi è un *remake*) il suo originale. Questa operazione ha come precondizione che l'anno di uscita di f sia minore o uguale all'anno di uscita del film stesso.

Le operazioni della classe Distributore sono le seguenti.

InserisciFilm($f : Film, c : unsigned$)
Il film f viene inserito nel catalogo del distributore con c copie (tutte disponibili). Se il film f è già in catalogo, le sue copie (totali e presenti) vengono aumentate di c unità.

EliminaFilm($f : Film$)
Il film f viene eliminato dal catalogo del distributore.
Precondizioni: Il film f è in catalogo nel distributore e tutte le copie sono presenti.

EsceFilm($f : Film$)
Una copia del film f viene data in prestito.
Precondizioni: Il film f è in catalogo nel distributore e almeno una copia è presente.

RitornaFilm($f : Film$)
Una copia del film f torna dal prestito.
Precondizioni: Il film f è in catalogo nel distributore e almeno una copia non è presente.

Infine, per scelta in fase di specifica, gli *unici* selettori disponibili nella classe Distributore per l'ispezione del catalogo devono essere i seguenti.

CopieFilm($f : Film$) : $\langle t, p \rangle$
Se f è in catalogo restituisce la coppia composta dal numero di copie totali t e attualmente presenti p del film f, altrimenti restituisce la coppia $\langle 0, 0 \rangle$.

ListaFilmPerAnno($a : unsigned$) : $vector\langle Film \rangle$
Restituisce il vettore di tutti i film in catalogo usciti nell'anno a.

ListaRemake() : $vector\langle Film \rangle$
Restituisce il vettore di tutti i film che sono il *remake* di un altro film *anch'esso in catalogo nel distributore*.

Esercizio 1 (punti 20) Si scriva la definizione delle classi C++ che compongono l'applicazione e delle loro funzioni membro.

Esercizio 2 (punti 5) Si scriva la definizione dell'operatore di output per la classe `Distributore` che produca anche gli attributi dei film (ma non le informazione legate all'associazione *remake*).

Esercizio 3 (punti 5) Si scriva una funzione esterna che prende come parametri un oggetto d di classe `Distributore`, il nome di un regista r e due anni $a1$ e $a2$ (con $a1 \leq a2$), e restituisce il vettore di film diretti da r negli anni tra $a1$ e $a2$ (estremi inclusi), che non sono dei *remake*.

Compito del 16 aprile 2007

Si vuole progettare un'applicazione per la gestione di una stazione ferroviaria. I treni possono arrivare da due direzioni distinte: nord e sud, identificate da un carattere: `'N'` o `'S'`. La stazione è costituita da più binari, denotati da numeri interi progressivi (a partire da 1), ed in ogni binario possono essere presenti più treni accodati. Il numero di binari è fissato una volta per tutte, mentre non si vogliono mettere limiti al numero di treni che un binario più ospitare. Ciascun treno presente in stazione può partire per una qualsiasi delle due direzioni, con l'ovvio vincolo che non vi siano altri treni, sullo stesso binario, che ostacolino la partenza in quella direzione.

La specifica della classe Stazione è la seguente:

Arriva($t : Treno, b : intero, d : carattere$)
Il treno t arriva in stazione nel binario b dalla direzione d.
Precondizioni: Il treno t non è già presente in stazione.

Parte($t : Treno, d : carattere$)
Il treno t parte dalla stazione in direzione d.
Precondizioni: Il treno t è presente in stazione e ha la direzione d libera sul suo binario.

Binario($t : Treno$) : $intero$
Restituisce il binario in cui si trova il treno t. Se t non è in stazione resituisce 0.

QuantiDavanti($t : Treno, d : carattere$) : *intero*

Restituisce il numero di treni che sono davanti a t in direzione d, sul suo binario. Se la direzione è libera restituisce 0.

Precondizioni: Il treno t è presente in stazione.

Esercizio 1 (punti 22) Si scriva la definizione della classe C++ `Stazione` e delle sue funzioni facendo riferimento alla classe `Treno` già sviluppata. Per la classe `Treno` si assuma disponibile soltanto l'operatore "`==`", ma non il costruttore di copia, né l'operatore di assegnazione, né il costruttore senza argomenti.

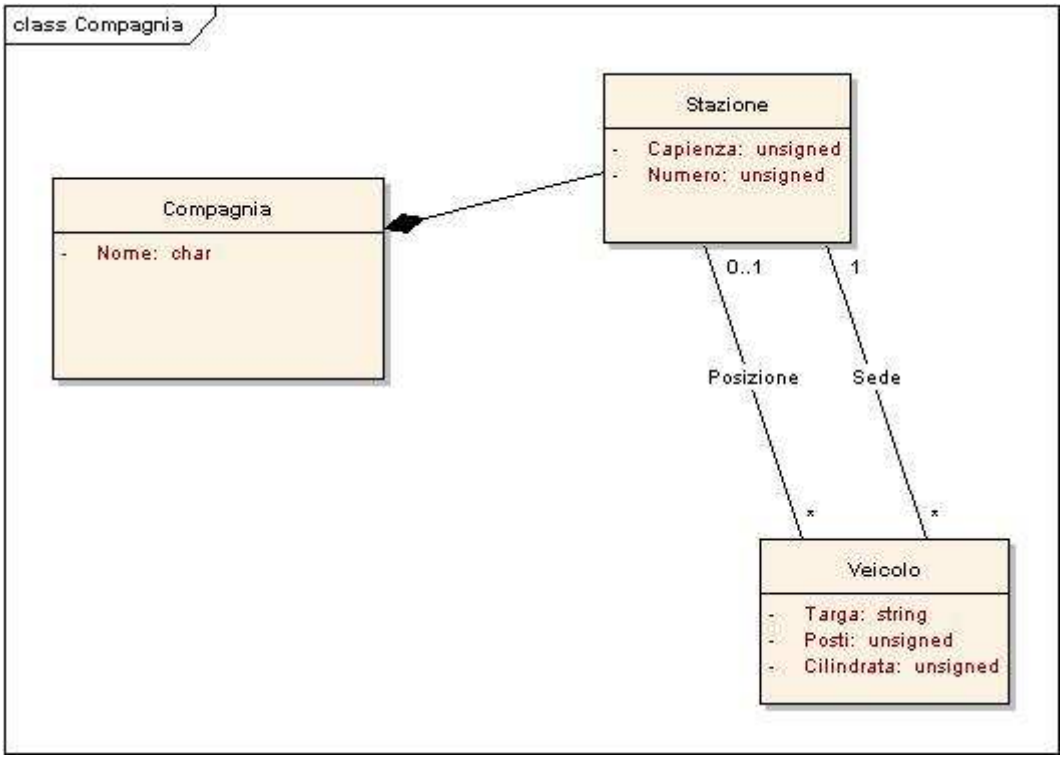
Esercizio 2 (punti 8) Nella stazione a volte è necessario spostare un treno da un binario ad un altro, e ciò viene fatto facendo "partire" tale treno dal binario dove si trova, e facendolo "arrivare" nel binario desiderato. La manovra è particolarmente complicata nel caso in cui il treno in questione sia fisicamente boccato da altri treni sullo stesso binario. In tal caso è necessario spostare altri treni, e si desidera ovviamente spostarne il minor numero possibile.

Si scriva una funzione esterna che prende come parametri una stazione, un treno ed un binario, e modifichi la stazione spostando il treno (già presente in stazione) nel binario passato come parametro, lasciando gli altri treni nei loro binari (in ordine qualsiasi).

Compito del 26 giugno 2007

Si vuole progettare un'applicazione per la gestione di una compagnia di noleggio di veicoli. La compagnia (identificata dal nome) è composta da un insieme di stazioni (identificate da un numero progressivo e di capienza limitata). In ciascuna stazione ha sede un insieme di veicoli, ciascuno identificato dalla targa e caratterizzato dal numero di posti e dalla cilindrata.

Ciascun veicolo in un dato istante può trovarsi nelle sua sede o essere in uso da un cliente. Oppure, siccome i clienti possono restituire un veicolo in una stazione diversa dalla sua sede, questo può anche trovarsi in un'altra stazione della compagnia. L'applicazione è descritta dal seguente diagramma delle classi UML.



La specifica delle operazione della classe Compagnia è la seguente:

Disponibile($i : unsigned, c : unsigned, s : Stazione$) : $Veicolo$

Restituisce un veicolo della compagnia presente in s che abbia almeno i posti e c cilindrata. Se

disponibile, restituisce un veicolo che non ha sede in s. Qualora nessun veicolo sia disponibile restituisce un valore fittizio (ad esempio un puntatore `NULL`).

Affitta($v : Veicolo, s : Stazione$)
Il veicolo v che si trova nella stazione s viene dato in affitto.
Precondizioni: Il veicolo v è disponibile in s.

Rientra($v : Veicolo, s : Stazione$)
Il veicolo v rientra nella stazione s.
Precondizioni: Il veicolo v è in uso ad un cliente e il numero di veicoli in s è inferiore alla sua capienza.

Riposiziona($v : Veicolo$)
Il veicolo v viene spostato dalla stazione dove si trova alla sua sede.
Precondizioni: Il veicolo v è di proprietà della compagnia e non è in uso ad un cliente e il numero di veicoli in s è inferiore alla sua capienza.

Aggiungi($v : Veicolo, s : Stazione$)
Il veicolo v viene aggiunto al parco macchine della compagnia con sede in s ed è presente in s.
Precondizioni: Il veicolo v non è già di proprietà della compagnia e il numero di veicoli in s è inferiore alla sua capienza.

Esercizio 1 (punti 24) Si scriva la definizione delle classi C++ `Compagnia` e `Stazione` e delle loro funzioni facendo riferimento alla seguente classe `Veicolo`.

```
class Veicolo
{
  friend bool operator==(const Veicolo& v1, const Veicolo& v2);
 public:
  Veicolo(string t, unsigned p, unsigned c)
    : targa(t) { posti = p; cilindrata = c; }
  string Targa() const { return targa; }
  unsigned Posti() const { return posti; }
  unsigned Cilindrata() const { return cilindrata; }
  bool RequisitiMinimi(unsigned p, unsigned c)
    { return posti <= p && cilindrata <= c; }
 private:
  string targa;
  unsigned posti;
  unsigned cilindrata;
};

inline bool operator==(const Veicolo& v1, const Veicolo& v2)
{ return v1.targa == v2.targa; }
```

Esercizio 2 (punti 6) Si scriva una funzione esterna che prende come parametro una compagnia e restituisce una coppia formata dalla percentuale di veicoli in uso ad utenti e la percentuale che si trova in una stazione diversa dalla sua sede.

Compito del 10 settembre 2007

Si vuole progettare un'applicazione per la gestione dei voli giornalieri di un aeroporto. I dati di interesse riguardano i voli in partenza dall'aeroporto, per i quali interessa il codice del volo, la compagnia aerea che lo opera, il nome del velivolo, l'orario previsto di partenza. Inoltre, per i voli interessa tener traccia anche degli eventuali ritardi, memorizzando l'orario effettivo di partenza, ma anche ciascun annuncio di ritardo con il nuovo orario previsto e l'orario in cui è stato dato l'annuncio (per un volo possono essere dati anche più annunci). Delle compagnie aeree occorre tener traccia del nome, della sede e del numero di velivolo posseduti.

Le operazioni fondamentali dell'aeroporto riguardano l'inserimento di un volo (passando non un oggetto del tipo corrispondente, bensì il codice, la compagnia, e l'orario previsto di partenza come parametri), la cancellazione di un volo, l'inserimento di un annuncio di ritardo e l'inserimento dell'orario effettivo di partenza.

Si consideri già disponibile la classe `Orario`, per la memorizzazione di un orario (ore e minuti), dotata di tutte le funzioni, i costruttori e gli operatori che si ritengono opportuni (ad esempio "`+`","`-`", "`<`").

Esercizio 1 (punti 6) Si tracci il diagramma UML dell'applicazione.

Esercizio 2 (punti 6) Si scriva la definizione delle classi C++ presenti nel diagramma UML.

Esercizio 3 (punti 14) Si scriva la definizione delle funzioni delle classi del diagramma UML (omettendo eventualmente quelle che si considerano meno rilevanti per l'esame).

Esercizio 4 (punti 4) Si scriva una funzione esterna che prende come parametro un aeroporto e restituisce il ritardo medio in minuti dei suoi voli.

Compito del 10 dicembre 2007 (soluzione a pagina 177)

Si vuole progettare un'applicazione per la gestione delle prenotazioni per uno spettacolo (con attributo nome). Ogni prenotazione riguarda un numero di posti ed è identificata da un codice intero. I posti sono numerati, e ciascun posto è identificato da due interi rappresentanti la fila e la colonna (la sala è rettangolare e i valori partono da 0). Le operazioni fondamentali della classe Spettacolo sono:

AggiungiPrenotazione(n : *intero*) : *intero*
: Inserisce nel sistema una prenotazione per un numero di posti pari a n. I posti assegnati sono scelti scandendo la sala per file partendo dal posto (0,0) e prendendo i primi n posti liberi. L'operazione restituisce il codice della prenotazione, il quale viene generato automaticamente dalla classe (in modo progressivo).

  Nel caso in cui non vi siano posti liberi a sufficienza, l'operazione restituisce -1 e nessuna prenotazione viene inserita.

AggiungiPosto(p : *Posto*) : *intero*
: Inserisce nel sistema una prenotazione per un singolo posto esattamente nella posizione p. L'operazione restituisce il codice della prenotazione, che viene generato dalla classe (come per *AggiungiPrenotazione*).

  Nel caso in cui il posto p sia già prenotato, l'operazione restituisce -1 e nessuna prenotazione viene generata.

RimuoviPrenotazione(c : *intero*) : *booleano*
: Rimuove la prenotazione corrispondente al codice c, libera tutti i posti associati e restituisce *true*. Se invece il codice c non esiste, l'operazione non ha effetto e restituisce $false$.

$VediPrenotazione(c : intero)$: $vettore\langle Posto\rangle$
: Restituisce il vettore di posti associato alla prenotazione di codice c. Se il codice c non esiste, restituisce un vettore vuoto.

Esercizio 1 (punti 5) Si tracci il diagramma UML dell'applicazione.

Esercizio 2 (punti 7) Si scriva la definizione delle classi C++, assumento che il nome e le dimensioni della sala siano forniti al costruttore della classe `Spettacolo`.

Esercizio 3 (punti 13) Si scriva la definizione delle funzioni delle classi.

Esercizio 4 (punti 5) Si scriva l'operatore di output della classe `Spettacolo` che fornisca l'output nel formato mostrato dal seguente esempio (codice eseguito a sinistra e output generato a destra):

```
int main()
{
  unsigned p1, p2, p3, p4, p5;
  Posto p(5,3);
  Spettacolo s("Amleto",7,5);
```

```
Sistema di prenotazione per lo spettacolo: Amleto

Mappa dei posti in sala:
XXXXX
XXXX-
```

```
  p1 = s.AggiungiPrenotazione(4);
  p2 = s.AggiungiPrenotazione(5);
  p3 = s.AggiungiPrenotazione(2);
  p4 = s.AggiungiPrenotazione(5);
  s.RimuoviPrenotazione(p3);
  p5 = s.AggiungiPosto(p);
  cout << s;
  return 0;
}
```

```
-XXXX
X----
-----
---X-
-----

Elenco prenotazioni:
Prenotazione 1 : (0,0) (0,1) (0,2) (0,3)
Prenotazione 2 : (0,4) (1,0) (1,1) (1,2) (1,3)
Prenotazione 4 : (2,1) (2,2) (2,3) (2,4) (3,0)
Prenotazione 5 : (5,3)
```

Compito del 21 dicembre 2007

Si vuole progettare un'applicazione per la gestione di un *circuito*. Un circuito ha una forma rettangolare ed è caratterizzato da lunghezza e larghezza (due valori interi positivi).

Nel circuito risiedono dei *componenti*, anch'essi rettangoli a dimensioni intere, identificati da una sigla alfanumerica. Ogni componente ha un insieme di *piedini*, che sono posizionati nel perimetro del rettangolo (in punti definibili da due coordinate intere), ma non nei vertici. Ciascun piedino ha un nome che non è modificabile.

Un componente risiede in un circuito in una data posizione, identificata dalle coordinate (anch'esse intere) relative sul circuito del suo vertice in alto a sinistra. Un componente non può essere ruotato, quindi il modo di inserirlo in una posizione è univoco.

Le operazione della classe Componente sono l'inserimento e la rimozione dei piedini, con le precondizioni: (*i*) un piedino può essere inserito solo sul perimetro del componente (esclusi i vertici) e non in un punto interno, (*ii*) due piedini non possono trovarsi nello stesso punto, (*iii*), non ci possono essere due piedini con lo stesso nome.

Le operazioni fondamentali della classe Circuito invece sono le seguenti, in cui si fa riferimenti al dominio *Coordinate* che consiste in una coppia di interi.

AggiungiComponente($c : Componente, xy : Coordinate$)
: Inserisce nel circuito il componente c nella posizione xy.
Precondizioni: Il componente c non è già presente nel circuito. Il componente c non si sovrappone con altri componenti nel circuito (può però trovarsi a contatto con altri componenti).

RimuoviComponente($c : Componente$)
: Rimuove il componente c dal circuito
Precondizioni: Il componente c è presente nel circuito.

SpostaComponente($c : Componente, xy : Coordinate$) : *booleano*
: Sposta il componente dalla sua posizione corrente alla posizione xy e restituisce *true*. Se però il componente c nella nuova posizione si sovrappone con altri componenti, allora lo spostamento non avviene e l'operazione restituisce $false$.

SostituisciComponente($c1 : Componente, c2 : Componente$) : *booleano*
: Il componente $c1$ viene sostituito dal componente $c2$ nella stessa posizione nel circuito. Se il componente $c2$ si sovrappone con altri componenti nel circuito allora la sostituzione non avviene e l'operazione restituisce $false$.

Esercizio 1 (punti 4) Si tracci il diagramma UML dell'applicazione.

Esercizio 2 (punti 6) Si scriva la definizione delle classi C++.

Esercizio 3 (punti 12) Si scriva la definizione delle funzioni delle classi.

Esercizio 4 (punti 4) Si scriva la definizione dell'operatore "`==`" per la classe `Circuito` che verifica se due circuiti sono esattamente uguali (dimensione, componenti, piedini e posizioni).

Esercizio 5 (punti 4) Si scriva una funzione booleana esterna (non *friend*) che prenda un oggetto di tipo `Circuito` e verifichi se ci sono almeno due piedini nel circuito che si trovano a contatto tra loro (cioè nella stessa posizione).

Compito del 17 marzo 2008

L'applicazione da progettare riguarda il gioco *Battaglia navale*. Il gioco è composto da una scacchiera quadrata $n \times n$ e da un insieme di *navi* posizionate sulla scacchiera. Ciascuna nave ha una dimensione k e occupa una sequenza di lunghezza k (orizzontale o verticale) di caselle della scacchiera adiacenti tra loro per un lato. Una casella non può essere occupata da due navi.

In un dato istante del gioco, ciascuna casella può essere stata *colpita* oppure no. Una nave che abbia almeno una casella colpita si dice *danneggiata*, mentre una nave che ha tutte le caselle colpite si dice *affondata*.

Come esempio si consideri la seguente scacchiera 6×6 con tre navi di lunghezza rispettivamente 4, 3 e 2, in cui il simbolo $\circ$ rappresenta una casella con una nave non colpita, il simbolo $\bullet$ una casella con una nave colpita, lo spazio bianco una casella vuota non colpita e la $\times$ una casella vuota colpita.

	×				
●	●	○			
		×		×	×
	○				
	○		×		
×		●	●	●	●

Nell'esempio la nave di lunghezza 4 è affondata, quella di lunghezza 3 è danneggiata e quella di lunghezza 2 è integra.

La classe++ C da progettare deve comprendere:

- Un costruttore che riceva come parametri la dimensione della scacchiera e la lista delle navi da inserire (nel modo che si ritiene più opportuno) e crei la scacchiera posizionando le navi in modo casuale. Si utilizzi la funzione `Random(int a, int b)` che restituisce un valore casuale tra `a` e `b`, estremi inclusi.
- Una funzione che *spari* su una casella, modificando la scacchiera e restituendo il risultato: nessuna nave colpita, nave colpita ma non affondata, nave affondata.
- Un insieme di selettori che permettano di sapere per ciascuna nave la lunghezza, la posizione e lo stato corrente (nel modo che si ritiene opportuno). Non è invece previsto un selettore che restituisca cosa è presente in una casella specifica.

Esercizio 1 (punti 10) Si scriva la definizione della classe `BattagliaNavale`.

Esercizio 2 (punti 10) Si scriva la definizione delle funzioni delle classe `BattagliaNavale`.

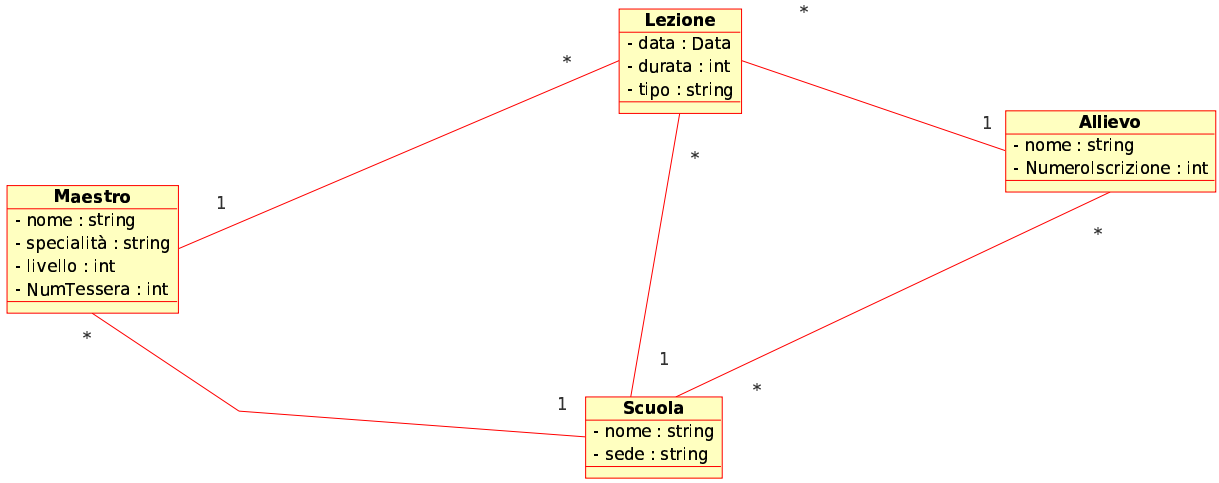
Esercizio 3 (punti 5) Si scriva la definizione (*friend*) dell'operatore di output per la classe `BattagliaNavale` che scriva i dati nel formato che si ritiene opportuno.

Esercizio 4 (punti 5) Si scriva una funzione esterna alla classe che prenda come parametri due oggetti di classe `BattagliaNavale` e verifichi se i due oggetti sono uguali a meno dei colpi sparati non sulle navi. Verifica cioè se la posizione e lo stato di danneggiamento/affondamento delle navi è identico (stesse caslle colpite).

Compito del 26 giugno 2008

Si vuole progettare una classe per la gestione delle lezioni di una scuola di sci. Ciascuna scuola è caratterizzata da un insieme di maestri, un insieme di allievi, e le lezioni impartite dai maestri agli allievi.

L'applicazione è descritta dal seguente schema in UML, senza operazioni. Le operazioni della classe Scuola sono specificate di seguito; le classi Maestro e Allievo non hanno operazioni; le operazioni della classe Lezione sono state omesse.



Le funzioni della classe Scuola sono specificate come segue:

InserisciAllievo(a : *Allievo*) : L'allievo a viene inserito nella scuola.
Precondizioni: il numero di iscrizione di a non è già presente nella scuola.

InserisciMaestro(m : *Maestro*) : Il maestro m viene inserito nella scuola. Se m è già presente allora l'operazione non ha effetto.
Precondizioni: non esiste già nella scuola un maestro con lo stesso numero di tessera ed un *altro* nome.

InserisciLezione(l : *Lezione*) : La lezione l viene inserita tra le lezioni effettuate.
Precondizioni: il maestro e l'allievo di l sono già presenti nella scuola; non è già presente una lezione dello stesso maestro allo stesso allievo nella stessa data e dello stesso tipo.

CreaLezione(m : *Maestro*, a : *Allievo*, d : *Data*) : Viene inserita una lezione di m ad a di durata 60', di tipo uguale alla specialità di m. Se il maestro m o l'allievo a dovessero non essere presenti nella scuola, allora vengono aggiunti.
Precondizioni: non è già presente una lezione di m ad a nella data d e dello stesso tipo.

Si assuma che non possano esistere maestri che abbiano lo stesso nome o le stesso numero di tessera e che non possano esistere assistiti che abbiano lo stesso numero di iscrizione.

Esercizio 1 (punti 10) Si scrivano le definizioni delle classi C++ `Scuola` e `Lezione` avendo già disponibili le classi `Maestro`, `Allievo` e `Data`. Le definizioni delle classi `Maestro` e `Allievo` sono quelle fornite di seguito, in cui alcune definizioni di funzione sono omesse per brevità (ma possono essere assunte già realizzate).
Si consideri disponibile la classe `Data` con tutte le funzioni di gestione che si ritengono necessarie.

Esercizio 2 (punti 10) Si scrivano le definizioni delle funzioni delle due classi, gestendo le precondizioni attraverso la funzione `assert()`.

Esercizio 3 (punti 6) Si scriva una funzione esterna che prenda come parametro una scuola e restituisca il nome del maestro di livello almeno 3 che ha il massimo rapporto tra il numero di lezioni di tipo corrispondente alla sua specialità ed il numero di lezioni totali che ha effettuato.

Esercizio 4 (punti 4) Si scriva l'operatore "`<<`" per la classe `Scuola` in modo che stampi tutte le informazioni ad essa associate.

Definizione delle classi già disponibili:

```
class Maestro
{
  friend ostream& operator<<(ostream& os, const Maestro& m);
public:
  Maestro(string n, string s, int l);
  string Nome() const {return nome; }
  string Specialita() const {return specialita; }
  int Livello() const { return livello; }
private:
```

```
  string nome, specialita;
  int livello;
};

class Allievo
{
  friend ostream& operator<<(ostream& os, const Allievo& a);
public:
  Allievo(string n, int ni);
  string Nome() const {return nome; }
  int NumIscrizione() const { return num_iscrizione; }
private:
  string nome;
  int num_iscrizione;
};
```

Soluzioni

Soluzione compito del 3 aprile 2001

Per rappresentare la scacchiera e la sua associazione con i pezzi, si usa una matrice di puntatori ai pezzi. Utilizzando la STL, la matrice è implementata tramite un vettore di vettori.

Siccome le coordinate della scacchiera partono da `1`, mentre gli indici dei vettori in C++ partono da `0`, le funzioni della classe `Scacchiera` che ricevono come parametri le coordinate `i` e `j` agiscono sulle locazioni `i-1` e `j-1`.

Si noti la presenza della funzione `Occupante()`, non menzionata nella specifica, ma che rappresenta un selettore necessario per poter ispezionare la posizione sulla scacchiera.

Come per altri esercizi, la classe `Pezzo` viene simulata tramite uno *stub*.

Soluzione esercizio 1

```
// File Scacchiera.h
#ifndef SCACCHIERA_H
#define SCACCHIERA_H

#include <vector>
#include <string>
#include <iostream>
#include "Pezzo.h"

class Scacchiera
{friend ostream& operator<<(ostream& os, const Scacchiera&);
public:
  Scacchiera(unsigned n);
  void Piazzamento(Pezzo* p, unsigned i, unsigned j);
  int Spostamento(unsigned i, unsigned j, unsigned d);
  unsigned Dim() const { return s.size(); }
  Pezzo* Occupante(unsigned i, unsigned j) const { return s[i-1][j-1]; }
private:
  vector<vector<Pezzo*> > s;
};
#endif
```

La funzione più complessa della classe `Scacchiera` è `Spostamento()`. Questa funzione utilizza due variabili per memorizzare la nuova posizione del pezzo. Prima di eseguire lo spostamento la funzione controlla se la mossa è eseguibile; in caso contrario, viene riconosciuta la causa e restituito il valore corrispondente.

Soluzione esercizio 2

```
// File Scacchiera.cpp
#include "Scacchiera.h"

Scacchiera::Scacchiera(unsigned n)
  : s(n)
```

```
{
  for (unsigned i = 0; i < s.size(); i++)
    s[i].resize(n);
}

void Scacchiera::Piazzamento(Pezzo* p, unsigned i, unsigned j)
{
  if (i >= 1 && i <= Dim() && j >= 1 && j <= Dim())
    s[i-1][j-1] = p;
}

int Scacchiera::Spostamento(unsigned i, unsigned j, unsigned d)
{
  unsigned ni = i, nj = j; // ni: nuova riga, nj: nuova colonna

  if (s[i-1][j-1] == NULL)
    return 0;
  switch (d)
    {
    case 1:  ni--; break;
    case 2:  nj++; break;
    case 3:  ni++; break;
    case 4:  nj--; break;
    }
  if (ni < 1 || ni > Dim() || j < 1 || j > Dim())
    return -2;
  if (s[ni-1][nj-1] != NULL)
    return -1;
  s[ni-1][nj-1] = s[i-1][j-1]; //sposta il pezzo
  s[i-1][j-1] = NULL;
  return 1;
}

ostream& operator<<(ostream& os, const Scacchiera& sc)
{
  for (unsigned i = 0; i < sc.Dim(); i++)
    {
      for (unsigned j = 0; j < sc.Dim(); j++)
        if (sc.s[i][j] != NULL)
          cout << sc.s[i][j]->Nome() << " ";
        else
          cout << "  ";
      cout << endl;
    }
  return os;
}
```

Soluzione esercizio 3 + driver

Per la funzione `PortaTuttiANord()` che risolve l'esercizio 3, esistono due algoritmi di soluzione diversi, ugualmente interessanti. Il primo è basato sull'idea di muovere un pezzo alla volta fino alla sua posizione finale, per passare poi al pezzo successivo. Il secondo sposta iterativamente tutti i pezzi a nord di una casella fino a che non sono tutti fermi. A scopo didattico, forniamo entrambe le soluzioni.

```
// File main.cpp

#include "Scacchiera.h"

void  PortaTuttiANord(Scacchiera& s);
void  PortaTuttiANord2(Scacchiera& s);
```

```
int main()
{
  int scelta, n, i, j, d;
  Pezzo* p_pezzo;
  string nome_pezzo;

  cout << "Dimensione scacchiera : ";
  cin >> n;
  Scacchiera s(n);

  do
    { cout << s;
      cout << "Quale operazione vuoi effettuare?\n"
           << "1: Piazzamento\n"
           << "2: Spostamento\n"
           << "3: Porta tutti a Nord\n"
           << "0: Esci\n\n"
           << "Scelta:  ";
      cin >> scelta;
      cout << endl;
      switch (scelta)
        {
        case 1:
          cout << "Quale pezzo : ";
          cin >> nome_pezzo;
          p_pezzo = new Pezzo(nome_pezzo);
          cout << "Quale casella : ";
          cin >> i >> j;
          s.Piazzamento(p_pezzo,i,j);
          break;
        case 2:
          {
            int ris;
            cout << "Quale casella : ";
            cin >> i >> j;
            cout << "Quale direzione (1:nord, 2:est, 3:sud, 4:ovest): ";
            cin >> d;
            ris = s.Spostamento(i,j,d);
            switch (ris)
              {
              case 1: cout << "Spostamento effettuato" << endl; break;
              case 0: cout << "Nessun pezzo da spostare" << endl; break;
              case -1: cout << "Casella di arrivo occupata" << endl; break;
              case -2: cout << "Casella di arrivo inesistente" << endl; break;
              }
          break;
          }
        case 3:
          PortaTuttiANord(s);
          break;
        }
    }
  while (scelta != 0);
}

// prima versione: sposta un pezzo alla volta
void PortaTuttiANord(Scacchiera &s)
{
```

```
  for (unsigned i = 2; i <= s.Dim(); i++)
    for (unsigned j = 1; j <= s.Dim(); j++)
      if (s.Occupante(i,j) != NULL)
        {
          unsigned k = i;
          while (s.Spostamento(k,j,1) == 1)
            k--;
        }
}

// seconda versione: sposta tutti i pezzi a ciascun giro
void PortaTuttiANord2(Scacchiera &s)
{
  bool tutti_fermi;
  do
    {
      tutti_fermi = true;
      for (unsigned i = 2; i <= s.Dim(); i++)
        for (unsigned j = 1; j <= s.Dim(); j++)
          {
            if (s.Spostamento(i,j,1) == 1)
              tutti_fermi = false;
          }
    }
  while (!tutti_fermi);
}
```

Classi *stub*

```
// File Pezzo.h
#ifndef PEZZO_H
#define PEZZO_H

using namespace std;

class Pezzo
{
public:
  Pezzo(string n) { nome = n; }
  string Nome() const { return nome; }
  bool operator==(const Pezzo& p) { return nome == p.nome; }
protected:
  string nome;
};
#endif
```

Soluzione compito del 5 luglio 2001

Dalle specifiche si deduce che la responsabilità della associazione tra la classe Appartamento e la classe Stanza deve essere della classe Appartamento. Ne consegue che la classe `Appartamento` dovrà avere tra i suoi dati un vettore di puntatori a `Stanza`. Analogamente, la classe `Stanza` avrà come dato un puntatore ad un oggetto di classe `Colore`.

Soluzione esercizio 1

```
// File appartamenti.h
#ifndef APPARTAMENTI_H
```

```
#define APPARTAMENTI_H

#include <vector>
#include <string>
#include <iostream>

using namespace std;

class Colore
{
public:
  Colore(string d);
  string Descrizione() const { return descrizione; }
private:
  string descrizione;
};

class Stanza
{
public:
  Stanza(string t);
  string Tipo() const { return tipo; }
  Colore* QualeColore() const { return colore; }
  void FissaColore(Colore* c);
private:
  string tipo;
  Colore* colore;
};

class Appartamento
{
  // operatore << non richiesto per il compito
  friend ostream& operator<<(ostream& os, const Appartamento& a);
public:
  Appartamento(string n);
  void AggiungiStanza(Stanza* s);
  Stanza* QualeStanza(unsigned i) const;
  int NumeroStanze() const { return stanze.size(); }
private:
  string nome;
  vector<Stanza*> stanze;
};

#endif
```

Soluzione esercizio 2

```
// File appartamenti.cpp

#include "appartamenti.h"
#include <cassert>

Colore::Colore(string d)
{
  descrizione = d;
}

Stanza::Stanza(string t)
{
```

```
  tipo = t;
  colore = NULL;
}

void Stanza::FissaColore(Colore* c)
{
  colore = c;
}

Appartamento::Appartamento(string n)
{
  nome = n;
}

void Appartamento::AggiungiStanza(Stanza* s)
{
  stanze.push_back(s);
}

Stanza* Appartamento::QualeStanza(unsigned i) const
{
  assert (i <= stanze.size());
  return stanze[i];
}

ostream& operator<<(ostream& os, const Appartamento& a)
{
  Stanza* s;
  for (int i = 0; i < a.NumeroStanze(); i++)
    {
       s = a.QualeStanza(i);
       os << "Stanza " << i+1 << ": tipo " << s->Tipo()
          << ", colore ";
       if (s->QualeColore() != NULL)
         os << a.QualeStanza(i)->QualeColore()->Descrizione() ;
       else
         os << "ignoto";
       os << endl;
     }
   return os;
}
```

Per semplicità, invece di un driver (`main`) a menù abbiamo realizzato un driver che effettua semplicemente alcune chiamate alle funzioni di classe. Questo driver ovviamente permette un test meno accurato di un menù.

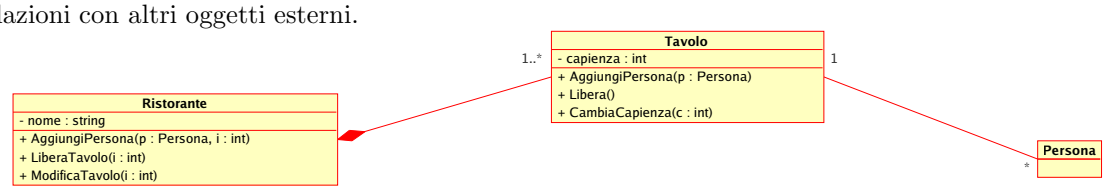
Soluzione esercizio 3 + driver

```
// File main.cpp

#include "appartamenti.h"

void FissaColore(Appartamento *a, Colore *c);

int main()
{
  Appartamento a("Casa Bianca");
  Colore c1("bianco");
  Colore c2("verde");
```

```
  Stanza s1("cucina");
  Stanza s2("bagno");
  Stanza s3("studio ovale");

  s1.FissaColore(&c2);
  a.AggiungiStanza(&s1);
  a.AggiungiStanza(&s2);
  a.AggiungiStanza(&s3);
  cout << a << endl;
  FissaColore(&a,&c1);
  cout << a << endl;
}

void FissaColore(Appartamento *a, Colore *c)
{
  for (int i = 0; i < a->NumeroStanze(); i++)
    a->QualeStanza(i)->FissaColore(c);
}
```

Soluzione compito del 23 luglio 2001

Anche se non richiesto dal testo è utile tracciare il diagramma UML dell'applicazione. Si noti che si è assunto che tra Ristorante e Tavolo vi sia una relazione di composizione (part-of), e che quindi la classe Tavolo abbia significato solo all'interno della classe Ristorante e che gli oggetti utilizzati non abbiano relazioni con altri oggetti esterni.



Come è facile evincere dalle specifiche, le responsabilità delle associazioni saranno nella classe a sinistra.

Come in altri esercizi, la classe `Persona` viene simulata da uno *stub*. Per questa classe è necessario l'operatore di uguaglianza "`==`" per poter verificare la presenza di una persona nel ristorante.

Essendo la relazione tra Ristorante e Tavolo di tipo part-of è consentito (e auspicabile, per semplicità) usare un vettore di oggetti (e non di puntatori).

Soluzione esercizio 1

```
// File ristorante.h
#ifndef RISTORANTE_H
#define RISTORANTE_H

#include <vector>
#include <iostream>
#include "persona.h"

class Tavolo
{
public:
  Tavolo(int posti = 4);
  int NumPosti() const { return capienza; }
  int NumSeduti() const { return seduti.size(); }
  bool Presente(Persona*) const;
  void AggiungiPersona(Persona*);
  void Libera();
```

```
  void CambiaCapienza(int c);
  Persona* Occupante(int i) const { return seduti[i]; }
private:
  vector<Persona*> seduti;
  int capienza;
};

class Ristorante
{
  friend ostream& operator<<(ostream& os, const Ristorante& r);
public:
  Ristorante(string n, int t);
  void AggiungiPersona(Persona* p, int i);
  void LiberaTavolo(int i);
  void ModificaTavolo(int i, int d);
  int NumTavoli() const { return tavoli.size(); }
  string Nome() const { return nome; }
  int PostiTavolo(int i) const { return tavoli[i].NumPosti(); }
  int SedutiTavolo(int i) const { return tavoli[i].NumSeduti(); }
private:
  bool PersonaPresente(Persona* p);
  vector<Tavolo> tavoli;
  string nome;
};
#endif
```

Soluzione esercizio 2

```
// File ristorante.cpp

#include "ristorante.h"
#include <cassert>

Tavolo::Tavolo(int posti)
  : seduti(0)
{
  capienza = posti;
}

bool Tavolo::Presente(Persona* p) const
{
  for (int i = 0; i < seduti.size(); i++)
    if (*(seduti[i]) == *p)
      return true;
  return false;
}

void Tavolo::AggiungiPersona(Persona* p)
{
  assert(seduti.size() < capienza);
  seduti.push_back(p);
}

void Tavolo::Libera()
{
  seduti.clear();
}

void Tavolo::CambiaCapienza(int c)
```

```
{
  assert(seduti.size() == 0);
  capienza = c;
}

Ristorante::Ristorante(string n, int t)
  : tavoli(t), nome(n)
{}

void Ristorante::AggiungiPersona(Persona* p, int i)
{
  if (!PersonaPresente(p))
    tavoli[i].AggiungiPersona(p);
}

void Ristorante::LiberaTavolo(int i)
{
  tavoli[i].Libera();
}

void Ristorante::ModificaTavolo(int i, int d)
{
  tavoli[i].CambiaCapienza(d);
}

bool Ristorante::PersonaPresente(Persona* p)
{
  for (int i = 0; i < tavoli.size(); i++)
    if (tavoli[i].Presente(p))
      return true;
  return false;
}

ostream& operator<<(ostream& os, const Ristorante& r)
{
  os << "Ristorante " << r.nome << endl;
  for (int i = 0; i < r.tavoli.size(); i++)
    {
      os << "Tavolo " << i+1 << " (" << r.PostiTavolo(i) << " posti) : ";
      for (int j = 0; j < r.SedutiTavolo(i); j++)
        os << r.tavoli[i].Occupante(j)->Nome() << ' ';
      os << endl;
    }
  return os;
}
```

Soluzione esercizio 3 + driver

Esistono numerosi algoritmi per dividere i gruppi nei vari tavoli. Trovare la soluzione *ottima*, cioè quella che minimizza il numero di gruppi divisi su più tavoli, è un problema complesso che va oltre gli scopi di questo corso (si veda il corso di Ricerca Operativa).

In questa sede proponiamo due soluzioni, di complessità (e di qualità della soluzione) crescente. La prima semplicemente assegna ciascuna persona di un gruppo al primo posto libero partendo dai tavoli di indice minori (nota: questa soluzione è stata ritenuta totalmente accettabile in sede di esame).

La seconda soluzione invece si compone di due fasi. Nella prima si mettono i gruppi nei tavoli che li contengono, senza dividere alcun gruppo. Nella seconda fase si sistemano gli eventuali gruppi rimasti nei posti liberi residui.

```
// File main.cpp
```

```
#include "persona.h"
#include "ristorante.h"
#include <iostream>

void AssegnaGruppi(Ristorante* r, vector<Gruppo*> gr);
void AssegnaGruppi2(Ristorante* r, vector<Gruppo*> gr);

int main()
{
  Ristorante ris1("La pergola",7);
  Ristorante ris2("Da Piero",4);
  Persona p1("Mario");
  Persona p2("Luca");
  Persona p3("Francesca");
  Persona p4("Irene");
  Persona p5("Laura");
  Persona p6("Giulio");
  Persona p7("Ilaria");
  Persona p8("Giovanni");
  Persona p9("Alberto");
  Persona p10("Claudia");
  Persona p11("Ludovica");
  Persona p12("Paolo");

  ris1.AggiungiPersona(&p1,1);
  ris1.AggiungiPersona(&p2,2);
  ris1.AggiungiPersona(&p3,2);
  ris1.ModificaTavolo(3,10);
  cout << ris1;
  Gruppo g1,g2, g3, g4;
  g1.AggiungiPersona(&p1);
  g1.AggiungiPersona(&p2);
  g1.AggiungiPersona(&p3);
  g1.AggiungiPersona(&p4);
  g1.AggiungiPersona(&p5);
  g2.AggiungiPersona(&p6);
  g2.AggiungiPersona(&p7);
  g3.AggiungiPersona(&p8);
  g3.AggiungiPersona(&p9);
  g3.AggiungiPersona(&p10);
  g4.AggiungiPersona(&p11);
  g4.AggiungiPersona(&p12);
  ris2.ModificaTavolo(0,6);
  vector<Gruppo*> vet(4);
  vet[0] = &g1;
  vet[1] = &g2;
  vet[2] = &g3;
  vet[3] = &g4;
  AssegnaGruppi(&ris2,vet);
  cout << ris2;
}

void AssegnaGruppi(Ristorante* r, vector<Gruppo*> gr)
{// versione semplice: riempie i tavoli spezzando i gruppi
  int i, j, k = 0;
  for (i = 0; i < gr.size(); i++)
    for (j = 0; j < gr[i]->QuantePersone(); j++)
      {
        if (r->PostiTavolo(k) == r->SedutiTavolo(k))
```

```
          k++;
        r->AggiungiPersona(gr[i]->QualePersona(j),k);
      }
}

void AssegnaGruppi2(Ristorante* r, vector<Gruppo*> gr)
{ // versione 2: siede un gruppo per tavolo e poi completa
  // i tavoli con gli altri gruppi
  int i, j, k = 0;
  for (i = 0; i < gr.size() && k < r->NumTavoli(); i++)
    {
        for (j = 0; j < gr[i]->QuantePersone(); j++)
          {
            r->AggiungiPersona(gr[i]->QualePersona(j),k);
            if (r->PostiTavolo(k) == r->SedutiTavolo(k))
              k++;
          }
        k++;
    }
  if (i < gr.size())
    { // i gruppi non sono finiti
      k = 0;
      for (i = 0; i < gr.size(); i++)
        for (j = 0; j < gr[i]->QuantePersone(); j++)
          {
            while (r->PostiTavolo(k) == r->SedutiTavolo(k))
              k++;
            r->AggiungiPersona(gr[i]->QualePersona(j),k);
          }
    }
}
```

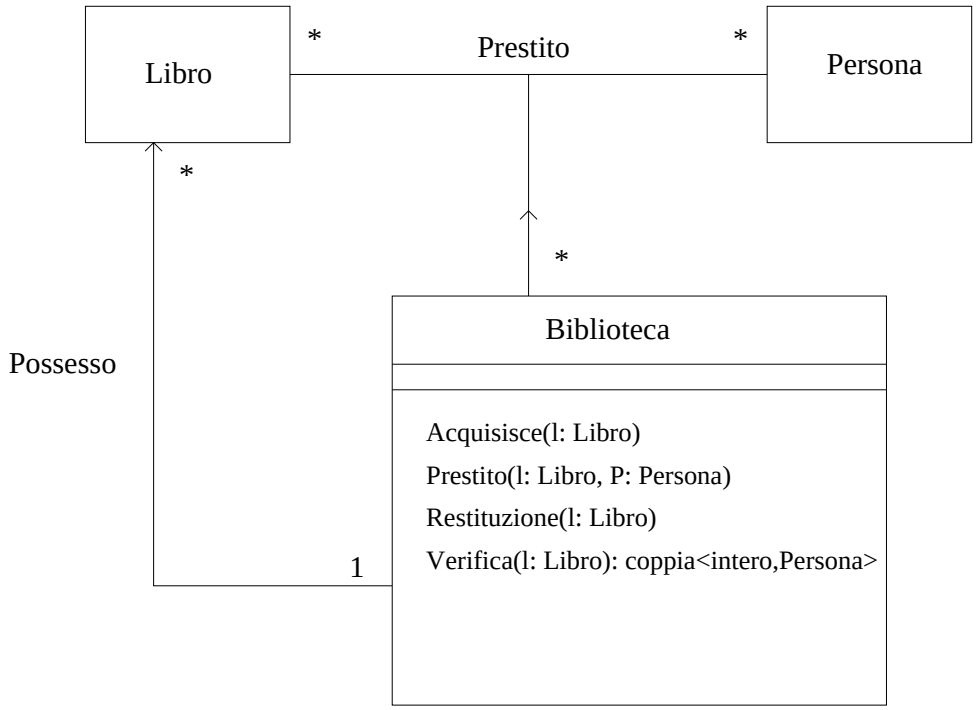
Classi predefinite

```
// File persona.h
#ifndef PERSONA_H
#define PERSONA_H

#include <string>
#include <vector>

using namespace std;

class Persona
{
public:
  Persona(string n) : nome(n) {}
  string Nome() const { return nome; }
  bool operator==(const Persona & p) { return nome == p.nome; }
private:
  string nome;
};

class Gruppo
{
 public:
  int QuantePersone() const { return membri.size(); }
  Persona* QualePersona(int i) { return membri[i]; }
```

```
  void AggiungiPersona(Persona* p) { membri.push_back(p); }
private:
  vector<Persona*> membri;
};
#endif
```

Soluzione compito del 13 settembre 2001

Anche se non richiesto dal compito, è utili traccare il diagramma diagramma UML dell'applicazione, in cui si evidenzia il fatto che la responsabilità di entrambe le associazioni ricade sulla classe Biblioteca.



Essendo evidente che la biblioteca non può prestare un libro che non possiede e che di conseguenza i libri in prestito sono un sottoinsieme dei libro posseduti, è possibile rappresentare entrambi le associazioni con un'unica struttura dati.

A questo scopo, utilizziamo una classe ausiliaria, chiamata `RecordLibro`, che contiene il puntatore al libro in oggetto, un campo booleano che indica se è in prestito, ed un puntatore ad una persona, che indica l'eventuale affidatario. Con questa scelta, la classe `Biblioteca` conterrà come dati unicamente un vettore di oggetti di tipo `RecordLibro`, corrispondente ai libri che possiede.

Come in altri esercizi, le classi `Persona` e `Libro` hanno delle implementazioni come *stub*. Per la classe `Libro`, assumiamo la presenza dell'operatore "`==`" indispensabile per poter cercare i libri nella biblioteca.

Soluzione esercizio 1

```
// file Biblioteca.h
#ifndef BIBLIOTECA_H
#define BIBLIOTECA_H

#include <string>
#include <vector>
#include <iostream>
#include "dati.h"

class RecordLibro
```

```
{
public:
  Libro* volume;
  Persona* affidatario;
  bool in_prestito;
};

class Biblioteca
{
  friend ostream& operator<<(ostream&, const Biblioteca& b);
public:
  Biblioteca();
  void Acquisisce(Libro *);
  void Prestito(Libro*, Persona*);
  void Restituzione(Libro*);
  unsigned Verifica(Libro*, Persona*&) const;
private:
  int CercaLibro(Libro* l) const;
  vector<RecordLibro> bib;
};
#endif
```

Soluzione esercizio 2

```
// file Biblioteca.cpp
#include "biblioteca.h"
#include <cassert>

Biblioteca::Biblioteca() {}

void Biblioteca::Acquisisce(Libro * l)
{
  RecordLibro r;
  r.volume = l;
  r.affidatario = NULL;
  r.in_prestito = false;
  bib.push_back(r);
}

int Biblioteca::CercaLibro(Libro* l) const
{
  for (unsigned i = 0; i < bib.size(); i++)
    if (bib[i].volume == l)
      return i;
  return -1;
}

void Biblioteca::Prestito(Libro* l, Persona* p)
{
  int i = CercaLibro(l);
  assert(i != -1);
  assert(!bib[i].in_prestito);
  bib[i].affidatario = p;
  bib[i].in_prestito = true;
}

void Biblioteca::Restituzione(Libro* l)
{
  int i = CercaLibro(l);
```

```
  assert(i != -1);
  assert(bib[i].in_prestito);
  bib[i].affidatario = NULL;
  bib[i].in_prestito = false;
}

unsigned Biblioteca::Verifica(Libro* l, Persona*& p) const
{
  int i = CercaLibro(l);
  if (i == -1)
    return 0;
  else
    if (bib[i].in_prestito)
      {
        p = bib[i].affidatario;
        return 2;
      }
    else
      return 1;
}

ostream& operator<<(ostream& os, const Biblioteca& b)
{
   for (unsigned i = 0; i < b.bib.size(); i++)
   {
      os << b.bib[i].volume->Nome() << " ";
      if (b.bib[i].in_prestito)
        os << "in prestito a " << b.bib[i].affidatario->Nome() << endl;
      else
        os << "disponibile" << endl;
   }
   return os;
}
```

Soluzione esercizio 3 + driver

```
// file main.cpp
#include <iostream>
#include <vector>
#include "biblioteca.h"

unsigned ControllaStato(const Biblioteca& b, const vector<Libro*>& v);

int main()
{
  Libro l1("a"), l2("b"), l3("c"), l4("d");
  Persona p1("p1");
  vector<Libro*> vet(3);
  Biblioteca b;

  vet[0] = &l1;
  vet[1] = &l2;
  vet[2] = &l3;
  b.Acquisisce(&l1);
  b.Acquisisce(&l2);
  b.Acquisisce(&l3);
  cout << b << endl;
  b.Prestito(&l1, &p1);
  b.Prestito(&l2, &p1);
```

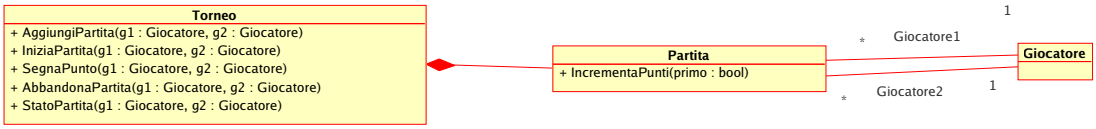
```
  cout << b << endl;
  cout << "Stato vettore : " <<  ControllaStato(b,vet) << endl;
  b.Restituzione(&l1);
  b.Restituzione(&l2);
  cout << b << endl;
  cout << "Stato vettore : " <<  ControllaStato(b,vet) << endl;
  vet[2] = &l4;
  cout << "Stato vettore : " <<  ControllaStato(b,vet) << endl;
}

unsigned ControllaStato(const Biblioteca& b, const vector<Libro*>& v)
{
  Persona* p;
  unsigned status;
  bool esiste_libro_in_prestito = false;
  for (unsigned i = 0; i < v.size(); i++)
  {
    status = b.Verifica(v[i],p);
    if (status == 0)
      return 0;
    else if (status == 2)
      esiste_libro_in_prestito = true;
  }
  if (esiste_libro_in_prestito)
    return 2;
  else
    return 1;
}
```

Classi *stub*

```
// file Dati.h
#ifndef DATI_H
#define DATI_H

#include <string>

using namespace std;

class Persona
{
public:
  Persona(string n) { nome = n; }
  string Nome() const { return nome; }
private:
  string nome;
};

class Libro
{
public:
  Libro(string n) { nome = n; }
  string Nome() const { return nome; }
  bool operator==(const Libro& l) { return nome == l.nome; }
private:
  string nome;
};
#endif
```

Soluzione compito del 27 settembre 2001

Definiamo innanzitutto lo schema UML dell'applicazione.



Le responsabilità delle relazioni sono assegnata alle classi Torneo e Partita rispettivamente, quindi la classe `Torneo` avrà come membro un vettore di partite.

Come specificato nel testo, ciascuna coppia di giocatori può incontrarsi al massimo una volta: dati due giocatori `g1` e `g2`, la partita tra `g1` e `g2` è la stessa della partita tra `g2` e `g1`. Questo significa che, per avere la necessaria generalità, la classe deve accettare che i parametri delle funzioni che gestiscono partite siano ordinati in modo arbitrario.

A questo scopo, le funzioni della classe `Torneo` si avvalgono di una funzione ausiliaria (privata), chiamata `CercaPartita()`, che dati due giocatori `g1` e `g2` restituisce non solo l'indice del vettore in cui si trova la partita `g1-g2`, ma anche un valore booleano. Questo valore sarà `true` se la partita è memorizzata come `g1-g2`, e `false` se è memorizzata come `g2-g1`. In base a questo valore, le funzioni che invocato `CercaPartita()` si comportano in modo diverso. Ad esempio, la funzione `SegnaPunto()`, utilizza una variabile booleana, chiamata `in_ordine`, a cui viene assegnato il valore riportato da `CercaPartita()`. Nel caso in cui`in_ordine` è `true`, la funzione assegna il punto al primo giocatore, nel caso opposto lo assegna al secondo.

Soluzione esercizio 1

```
// File Torneo.h
#ifndef TORNEO_H
#define TORNEO_H
#include <string>
#include <vector>
#include <iostream>
#include "Giocatore.h"

class Partita
{
  friend ostream& operator<<(ostream& os, const Partita& p);
 public:
  Partita(Giocatore* g1, Giocatore* g2);
  Giocatore* PrimoGiocatore() const { return gioc1; }
  Giocatore* SecondoGiocatore() const { return gioc2; }
  unsigned Stato() const { return stato; }
  void SetStato(unsigned s) { stato = s; }
  void IncrementaPunti(bool primo);

 private:
  Giocatore *gioc1, *gioc2;
  unsigned stato;
  unsigned punti1, punti2, set1, set2;
};

class Torneo
{
  friend ostream& operator<<(ostream& os, const Torneo& t);
public:
  void AggiungiPartita(Giocatore* g1, Giocatore *g2);
  void IniziaPartita(Giocatore* g1, Giocatore *g2);
  void SegnaPunto(Giocatore* g1, Giocatore *g2);
  void AbbandonaPartita(Giocatore* g1, Giocatore *g2);
  unsigned StatoPartita(Giocatore* g1, Giocatore *g2) const;
```

```
private:
  vector<Partita> partite;
  // CercaPartita restituisce -1 se la partita non esiste, e l'indice del
  // vettore dove si trova se esiste. ordine_giusto e' true se g1 e g2 sono
  // messi in questo ordine, false se la partita e' (g2, g1)
  int CercaPartita(Giocatore* g1, Giocatore *g2, bool& ordine_giusto) const;
};
#endif
```

Soluzione esercizio 2

```
// File Torneo.cpp

#include "Torneo.h"
#include <cassert>

Partita::Partita(Giocatore* g1, Giocatore* g2)
{
  gioc1 = g1;
  gioc2 = g2;
  stato = 3; // partita in programma
  punti1 = 0;
  punti2 = 0;
  set1 = 0;
  set2 = 0;
}

void Partita::IncrementaPunti(bool primo)
{
  if (primo)
    {
      punti1++;
      if (punti1 == 5)
        {
          punti1 = 0;
          punti2 = 0;
          set1++;
          if (set1 == 2)
            stato = 0;
        }
    }
  else
    {
      punti2++;
      if (punti2 == 5)
        {
          punti2 = 0;
          punti1 = 0;
          set2++;
          if (set2 == 2)
            stato = 1;
        }
    }
}

ostream& operator<<(ostream& os, const Partita& p)
{
  os << p.gioc1->Nome() << " - " << p.gioc2->Nome() << ": "
     << p.punti1 << "-" << p.punti2
```

```
     << " (" << p.set1 << "-" << p.set2 << ")";
  switch (p.stato)
    {
    case 0: os << " (vince " << p.gioc1->Nome() << ")"; break;
    case 1: os << " (vince " << p.gioc2->Nome() << ")"; break;
    case 2: os << " (in corso)"; break;
    case 3: os << " (in programma)"; break;
    }
  return os;
}

void Torneo::AggiungiPartita(Giocatore* g1, Giocatore *g2)
{
  Partita p(g1,g2);
  partite.push_back(p);
}

void Torneo::IniziaPartita(Giocatore* g1, Giocatore *g2)
{
  bool in_ordine;
  int i;

  i = CercaPartita(g1,g2,in_ordine);
  assert(partite[i].Stato() == 3);
  partite[i].SetStato(2);
}

void Torneo::SegnaPunto(Giocatore* g1, Giocatore *g2)
{
  bool in_ordine;
  int i;

  i = CercaPartita(g1,g2,in_ordine);
  assert(partite[i].Stato() == 2);

  partite[i].IncrementaPunti(in_ordine);
}

void Torneo::AbbandonaPartita(Giocatore* g1, Giocatore *g2)
{
  bool in_ordine;
  int i;

  i = CercaPartita(g1,g2,in_ordine);
  assert(partite[i].Stato() == 2);
  if (in_ordine)
    partite[i].SetStato(1);
  else
    partite[i].SetStato(0);
}

unsigned Torneo::StatoPartita(Giocatore* g1, Giocatore *g2) const
{
  bool in_ordine;
  int i;

  i = CercaPartita(g1,g2,in_ordine);

  if (i == -1)
```

```
    return 4;
  else if (partite[i].Stato() >= 2)
    return partite[i].Stato();
  else
    if (in_ordine)
      return partite[i].Stato();
    else
      return (partite[i].Stato() + 1) % 2; // inverte 0 e 1
}

int Torneo::CercaPartita(Giocatore* g1, Giocatore *g2, bool& ordine_giusto) const
{
  for (unsigned i = 0; i < partite.size(); i++)
    {
      if (*(partite[i].PrimoGiocatore()) == *g1
          && *(partite[i].SecondoGiocatore()) == *g2)
        {
          ordine_giusto = true;
          return i;
        }
      else if (*(partite[i].PrimoGiocatore()) == *g2
               && *(partite[i].SecondoGiocatore()) == *g1)
        {
          ordine_giusto = false;
          return i;
        }
    }
  return -1;
}

ostream& operator<<(ostream& os, const Torneo& t)
{
  os << "Torneo" << endl;
  for (unsigned i = 0; i < t.partite.size(); i++)
    os << t.partite[i] << endl;
  return os;
}
```

Soluzione esercizio 3 + driver

```
// File Torneo.cpp

#include "Torneo.h"
#include <cassert>

void InputGiocatori(unsigned& i1, unsigned& i2, const vector<Giocatore*> &v);
unsigned CercaNome(string nome, const vector<Giocatore*> &v);
unsigned ContaVittorie(const Torneo& t, Giocatore* g, const vector<Giocatore*>& v);

int main()
{
  Torneo t;
  vector<Giocatore*> v(5);
  string nome;
  v[0] = new Giocatore("mario");
  v[1] = new Giocatore("francesco");
  v[2] = new Giocatore("silvia");
  v[3] = new Giocatore("roberta");
  v[4] = new Giocatore("piero");
```

```
  unsigned scelta, i1, i2;

  do
    { cout << t;
      cout << "Quale operazione vuoi effettuare?\n"
           << "1: AggiungiPartita\n"
           << "2: IniziaPartita\n"
           << "3: SegnaPunto\n"
           << "4: AbbandonaPartita\n"
           << "5: StatoPartita\n"
           << "6: ContaVittorie\n"
           << "0: Esci\n\n"
           << "Scelta:  ";
      cin >> scelta;
      cout << endl;
      switch (scelta)
        {
        case 1:
          InputGiocatori(i1,i2,v);
          t.AggiungiPartita(v[i1],v[i2]);
          break;
        case 2:
          InputGiocatori(i1,i2,v);
          t.IniziaPartita(v[i1],v[i2]);
          break;
        case 3:
          InputGiocatori(i1,i2,v);
          t.SegnaPunto(v[i1],v[i2]);
          break;
        case 4:
          InputGiocatori(i1,i2,v);
          t.AbbandonaPartita(v[i1],v[i2]);
          break;
        case 5:
          {
            InputGiocatori(i1,i2,v);
            switch(t.StatoPartita(v[i1],v[i2]))
              {
              case 0: cout << "Ha vinto " << v[i1]->Nome(); break;
              case 1: cout << "Ha vinto " << v[i2]->Nome(); break;
              case 2: cout << "Partita in corso "; break;
              case 3: cout << "Partita in programma "; break;
              case 4: cout << "Partita inesistente "; break;
              }
            break;
          }
        case 6:
          cout << "Giocatore: ";
          cin >> nome;
          i1 = CercaNome(nome,v);
          cout << "Vittorie sugli altri " << ContaVittorie(t,v[i1],v) << endl;
          // in v c'e' anche il giocatore in questione,
          // ma questo e' ininfluente
        }
    }
  while (scelta != 0);
}
```

```
unsigned ContaVittorie(const Torneo& t, Giocatore* g, const vector<Giocatore*>& v)
{
  unsigned conta = 0;
  for (unsigned i = 0; i < v.size(); i++)
    if (t.StatoPartita(g,v[i]) == 0)
      conta++;
  return conta;
}

void InputGiocatori(unsigned& i1, unsigned& i2, const vector<Giocatore*> &v)
{
  string nome;
  cout << "Primo giocatore: ";
  cin >> nome;
  i1 = CercaNome(nome,v);
  cout << "Secondo giocatore: ";
  cin >> nome;
  i2 = CercaNome(nome,v);
}

unsigned CercaNome(string n, const vector<Giocatore*> &v)
{
  for (unsigned i = 0; i < v.size(); i++)
    if (n == v[i]->Nome())
      return i;
  cerr << "Giocatore sconosciuto";
  assert(false);
  return 0;
}
```

Classi *stub*

```
// File Giocatore.h
#ifndef GIOCATORE_H
#define GIOCATORE_H

#include <string>

using namespace std;

class Giocatore
{
  friend bool operator==(const Giocatore& g1, const Giocatore& g2);
public:
  Giocatore(string n) { nome = n; }
  string Nome() const { return nome; }
private:
  string nome;
};

inline bool operator==(const Giocatore& g1, const Giocatore& g2)
{ return g1.nome == g2.nome; }

#endif
```

Soluzione compito del 10 dicembre 2001

Per questo esercizio, è necessario definire una classe che raccolga gli attributi dell'associazione tra la classe `Agenzia` e la classe `Persona` oltre ad un puntatore alla classe `Persona`, poter rappresentare l'associazione all'interno della classe `Agenzia`.

Definiamo quindi una classe chiamata `Cliente` che contiene un puntatore alla persona, un carattere che denota il sesso, ed un *riferimento* all'eventuale persona selezionata. Questo riferimento può essere sia un puntatore alla persona che l'indice della locazione della persona all'interno dell'agenzia. Abbiamo preferito quest'ultima soluzione perché è più semplice da gestire.

L'associazione sarà quindi rappresentata da un vettore di oggetti di classe `Cliente`.

Soluzione esercizio 1

```
// file Agenzia.h
#ifndef AGENZIA_H
#define AGENZIA_H
#include <string>
#include <vector>
#include "Persona.h"

class Cliente
{
 public:
  Cliente(Persona* p, char s);
  Persona* persona;
  char sesso;
  int persona_selezionata; // -1 = nessuna selezione
};

class Agenzia
{
 public:
  Agenzia(string n) { nome = n; }
  void InserisciCliente(Persona* p, char sesso);
  void InserisciSelezione(Persona* p, Persona* ps);
  vector<Persona*> CoppieCreate() const;
 private:
  string nome;
  vector<Cliente> clienti;
  int CercaPersona(Persona* p) const;
};
#endif
```

La funzione `CreaCoppie()` restituisce il vettore delle coppie create. Questo vettore è ottenuto scorrendo il vettore dei clienti e verificando per ciascun cliente se ha selezionato una persona e se quest'ultima ha selezionato il cliente stesso. Per evitare che una coppia venga inserita due volte nel vettore, si esegue il controllo solo sui clienti di un sesso (`F` nel codice).

Soluzione esercizio 2

```
// file Agenzia.cpp
#include "Agenzia.h"
#include <cassert>

Cliente::Cliente(Persona* p, char s)
{
  persona = p;
  sesso = s;
  persona_selezionata = -1;
}
```

```
void Agenzia::InserisciCliente(Persona* p, char sesso)
{
  clienti.push_back(Cliente(p,sesso));
}

void Agenzia::InserisciSelezione(Persona* p, Persona* ps)
{
  int i, j;
  i = CercaPersona(p);
  j = CercaPersona(ps);
  assert (i != -1 && j != -1);
  assert(clienti[i].sesso != clienti[j].sesso);
  clienti[i].persona_selezionata = j;
}

int Agenzia::CercaPersona(Persona* p) const
{
  for (unsigned i = 0; i < clienti.size(); i++)
    if (*(clienti[i].persona) == *p)
      return i;
  return -1;
}

vector<Persona*> Agenzia::CoppieCreate() const
{
  vector<Persona*> coppie;
  for (int i = 0; i < clienti.size(); i++)
    {
      int j = clienti[i].persona_selezionata;
      if (j != -1 && clienti[j].persona_selezionata == i &&
          clienti[i].sesso == 'F')
        {
          coppie.push_back(clienti[i].persona);
          coppie.push_back(clienti[j].persona);
        }
    }
  return coppie;
}
```

Soluzione esercizio 3 + driver

```
// file Agenzia.cpp
#include "Agenzia.h"

unsigned ContaCoppieOver50(const vector<Agenzia*>& agenzie);

int main()
{
  Persona p1("Mario", 35), p2("Paolo",32),
    p3("Francesca",23), p4("Claudia",25),
    p5("Francesco",56), p6("Irene",51),
    p7("Susanna",60), p8("Leonardo",70),
    p9("Alfredo",52), p10("Maria Francesca",49);
  Agenzia ag1("Sposami subito"),
          ag2("Baci e abbracci");
  vector<Agenzia*> agenzie(2);
  agenzie[0] = &ag1;
  agenzie[1] = &ag2;
```

```
  ag1.InserisciCliente(&p1,'M');
  ag1.InserisciCliente(&p2,'M');
  ag1.InserisciCliente(&p3,'F');
  ag1.InserisciCliente(&p4,'F');
  ag1.InserisciSelezione(&p1,&p3);
  ag1.InserisciSelezione(&p3,&p2);
  ag2.InserisciCliente(&p5,'M');
  ag2.InserisciCliente(&p6,'F');
  ag2.InserisciCliente(&p7,'F');
  ag2.InserisciCliente(&p8,'M');
  ag2.InserisciCliente(&p9,'M');
  ag2.InserisciCliente(&p10,'F');
  ag2.InserisciSelezione(&p7,&p8);
  ag2.InserisciSelezione(&p8,&p7); // crea una coppia over 50
  ag2.InserisciSelezione(&p9,&p10);
  ag2.InserisciSelezione(&p10,&p9); // crea una coppia non over 50
  cout << "Num. coppie Over 50 create : " << ContaCoppieOver50(agenzie)  << endl;
}

unsigned ContaCoppieOver50(const vector<Agenzia*>& agenzie)
{
  unsigned conta = 0;
  vector<Persona*> coppie;

  for (unsigned i = 0; i < agenzie.size(); i++)
    {
      coppie = agenzie[i]->CoppieCreate();
      for (unsigned j = 0; j < coppie.size(); j+=2)
        if (coppie[j]->Eta() > 50 && coppie[j+1]->Eta() > 50)
          conta++;
    }
  return conta;
}
```

Classe predefinita

```
// file Persona.h
#ifndef PERSONA_H
#define PERSONA_H
#include <string>

class Persona
{
 public:
  Persona(string n, unsigned e) { nome = n; eta = e; }
  string Nome() const { return nome; }
  unsigned Eta() const { return eta; }
  bool operator==(const Persona& p) const { return nome == p.nome && eta == p.eta; }
 private:
  string nome;
  unsigned eta;
};
#endif
```

Soluzione compito del 7 gennaio 2002

Iniziamo l'esercizio analizzando i dati necessari alla classe `GiocoOca`:

- Una rappresentazione del tabellone: per questo usiamo semplicemente un vettore di interi il cui significato è quello descritto nella specifica della operazione *DefinisciTabellone*.
- Lo stato della partita: un vettore per memorizzare la posizione di ciascun giocatore, e tre interi per memorizzare rispettivamente il numero di giocatori, il giocatore di turno, e l'eventuale vincitore

Soluzione esercizio 1

```
// File GiocoOca.h
#ifndef GIOCO_OCA_H
#define GIOCO_OCA_H
#include <vector>
#include <fstream>

using namespace std;

class GiocoOca
{
  friend istream& operator>>(istream& is, GiocoOca& g);
  friend ostream& operator<<(ostream& os, const GiocoOca& g);
public:
  GiocoOca();
  GiocoOca(const vector<int>& t, unsigned k);
  void Ricomincia();
  void DefinisciTabellone(const vector<int>& t);
  void DefinisciGiocatori(unsigned k);
  void EseguiMossa(unsigned d);
  unsigned Vincitore() const { return vincitore; }
private:
  vector<int> tabellone; // la locazione 0 non e' utilizzata
  vector<unsigned> posizione_giocatore; // la locazione 0 non e' utilizzata
  unsigned num_giocatori;
  unsigned giocatore_di_turno;
  unsigned vincitore; // 0 = nessun vincitore, la partita e; in corso
};
#endif
```

Soluzione esercizi 2 e 4

Le funzioni della classe non richiedono particolari commenti. Si noti solo che nella funzione `EseguiMossa()` non c'è bisogno di distinguere tra caselle "Avanti", caselle "Indietro" e caselle normali, in quando tutte e tre si risolvono con la somma del valore della casella (positivo, negativo o nullo).

Gli operatori "<<" e ">>" scrivono e leggono lo stato di una partita. A questo scopo viene definito un formato testuale di tutti i dati della classe `GiocoOca` che viene interpretato dai due operatori. Il formato utilizzato è il seguente:

```
<num_caselle> <casella1> <casella2> ...
<num_giocatori> <pos_gioc1> <pos_gioc2> ...
```

Il dato sul vincitore non viene memorizzato perchè si assume che una partita venga letta o scritta solo se in corso.

```
// File GiocoOca.cpp
#include "GiocoOca.h"
#include <cassert>

GiocoOca::GiocoOca()
```

```
  // crea un tabellone ed un numero di giocatori di default:
  // 10 caselle tutte normali, 2 giocatori
  : tabellone(10,0), posizione_giocatore(3,1)
{
  num_giocatori = 2;
  giocatore_di_turno = 1;
  vincitore = 0;
}

GiocoOca::GiocoOca(const vector<int>& t, unsigned k)
  : tabellone(t), posizione_giocatore(k+1,1)
{
  num_giocatori = k;
  giocatore_di_turno = 1;
  vincitore = 0;
}

void GiocoOca::Ricomincia()
{
  for (unsigned i = 1; i <= num_giocatori; i++)
    posizione_giocatore[i] = 1;
  giocatore_di_turno = 1;
  vincitore = 0;
}

void GiocoOca::DefinisciTabellone(const vector<int>& t)
{
  tabellone = t;
  Ricomincia();
}

void GiocoOca::DefinisciGiocatori(unsigned k)
{
  posizione_giocatore.resize(k+1);
  Ricomincia();
}

void GiocoOca::EseguiMossa(unsigned d)
{
  assert(vincitore == 0);
  // esegui lancio
  posizione_giocatore[giocatore_di_turno] += d;
  if (posizione_giocatore[giocatore_di_turno] > tabellone.size())
    {
      vincitore = giocatore_di_turno;
      return;
    }
  // esegui istruzione
  posizione_giocatore[giocatore_di_turno] += tabellone[posizione_giocatore[giocatore_di_turno]];
  if (giocatore_di_turno == num_giocatori)
    giocatore_di_turno = 1;
  else
    giocatore_di_turno++;
}

istream& operator>>(istream& is, GiocoOca& g)
{
  // formato: num_caselle casella1 casella2 ...
  //          num_giocatori pos_gioc1 pos gioc2 ...
```

```
  //          giocatore di turno
  // assumiano che la partita sia in corso
  unsigned dim_tabellone, i;

  is >> dim_tabellone;
  g.tabellone.resize(dim_tabellone + 1);
  for (i = 1; i <= dim_tabellone; i++)
    is >> g.tabellone[i];
  is >> g.num_giocatori;
  for (i = 1; i <= g.num_giocatori; i++)
    is >> g.posizione_giocatore[i];
  is >> g.giocatore_di_turno;
  return is;
}

ostream& operator<<(ostream& os, const GiocoOca& g)
{
  unsigned i;
  os << g.tabellone.size() - 1;
  for (i = 1; i < g.tabellone.size(); i++)
    os << ' ' << g.tabellone[i];
  os << endl << g.num_giocatori;
  for (i = 1; i <= g.num_giocatori; i++)
    os << ' ' << g.posizione_giocatore[i];
  os << endl << g.giocatore_di_turno;
  return os;
}
```

Soluzione esercizio 3 + driver

La funzione esterna `PartitaGiocoOca()` semplicemente dichiara un oggetto della classe `GiocoOca`, ed esegue i lanci assegnati su quella partita. Si noti che questa funzione fa affidamento sul fatto che il costruttore della classe posiziona tutte le pedine nella casella iniziale.

```
// File main.cpp

#include <fstream>
#include <iostream>
#include <vector>
#include <string>
#include <cassert>
#include "GiocoOca.h"
#include "../Utilities/Random.h"

unsigned PartitaGiocoOca(const vector<int>& tab, unsigned gioc, const vector<unsigned> &lanci);
void ProvaPartitaGiocoOca();
void ProvaOperatoriIO();

int main()
{
  ProvaPartitaGiocoOca();
  ProvaOperatoriIO();
}

void ProvaPartitaGiocoOca()
{
  unsigned num_caselle, num_giocatori, i;
  vector<int> tabellone;
  vector<unsigned> lanci(100);
```

```
  cout << "Gioco dell'oca" << endl
       << "Numero di caselle del tabellone : ";
  cin >> num_caselle;
  tabellone.resize(num_caselle + 1);
  cout << "Tabellone : ";
  for (i = 1; i <= num_caselle; i++)
    cin >> tabellone[i];
  cout << "Numero giocatori: ";
  cin >> num_giocatori;

  cout << "Prova Funzione PartitaGiocoOca: Creo una sequenza di 100 lanci random\n";
  for (i = 0; i < lanci.size(); i++)
    lanci[i] = Random(1,6);

  cout << "Il vincitore delle partita e' il giocatore numero "
       << PartitaGiocoOca(tabellone, num_giocatori, lanci)
       << endl;
}

void ProvaOperatoriIO()
{
  cout << "Prova operatori <<  e >> " << endl;
  GiocoOca g;
  string nome_file;
  unsigned lancio;
  cout << "Nome file contenente una partita in corso : ";
  cin >> nome_file;
  ifstream is(nome_file.c_str());
  is >> g;
  cout << g << endl;
  cout << "Lancio : ";
  cin >> lancio;
  g.EseguiMossa(lancio);
  is.close();
  ofstream os(nome_file.c_str());
  os << g;
}

unsigned PartitaGiocoOca(const vector<int>& tab, unsigned gioc, const vector<unsigned> &lanci)
{
  GiocoOca g(tab, gioc);
  for (unsigned i = 0; i < lanci.size(); i++)
  {
    g.EseguiMossa(lanci[i]);
    if (g.Vincitore() != 0)
      return g.Vincitore();
  }
  assert(false); // la sequenza non e' bastata
}
```

Soluzione compito del 8 aprile 2002

Per rappresentare una scacchiera utilizziamo semplicemente una matrice quadrata di caratteri, che denotano i pezzi. Ciascun pezzo è quindi codificato con un singolo carattere. La codifica è la seguente: `'b'`: pedina bianca, `'n'` pedina nera, `'z'` zen, `' '` (spazio bianco) casella vuota.

Per identificare le caselle utilizziamo la classe `Casella` che raccoglie le due coordinate in un singolo oggetto. Per denotare le mosse, utilizziamo la casella di partenza e la direzione di movimento. Le direzioni sono denotate da una stringa nell'insieme `{"nord","sud","est","ovest","nord-est", ...}` degli otto punti cardinali.

Per denotare il giocatore di turno, cioè colui a cui spetta la prossima mossa, utilizziamo ancora un singolo carattere, con la seguente codifica: `'b'` bianco, `'n'` nero, `'x'` partita finita. Si noti che abbiamo usato lo stesso carattere, `'b'` o `'n'`, sia per i pezzi che per il giocatore di turno. Questa uguaglianza viene sfruttata quando ci chiediamo se una data casella è occupata da un pezzo del giocatore di turno, utilizzando semplicemente l'operatore "`==`" tra caratteri.

Per ispezionare la scacchiera ci affidiamo alla funzione `Pezzo()` che, data una casella, restituisce il carattere corrispondente al pezzo presente nella casella. Nel caso che la casella sia fuori dalla scacchiera, la funzione restituisce uno spazio bianco (cioè la casella vuota). Per comodità d'uso, forniamo due versioni simili della funzione `Pezzo()` che ricevono come parametri o un oggetto di tipo `Casella` oppure direttamente due interi.

Soluzione esercizio 1

```
// File ScacchiNibelunghi.h
#ifndef SCACCHI_NIBELUNGHI_H
#define SCACCHI_NIBELUNGHI_H
#include <vector>
#include <string>
#include <iostream>

using namespace std;

//                CODIFICHE
// - i pezzi sono codificati con un carattere:
//    b: pedina bianca, n: pedina nera, z: Zen, blank: casella vuota
// - le direzioni di mossa sono codificate con una stringa
//    "nord", "sud", "nordovest", ...
// - il giocatore di turno e' codificato con un carattere:
//    b: bianco, n: nero, x: partita finita
const int DIM = 5;

class Casella
{
public:
  Casella(int i, int j) { riga = i; colonna = j; }
  int riga, colonna;
};

class Partita
{
  friend ostream& operator<<(ostream& os, const Partita& g);
public:
  Partita();
  void RicominciaPartita();
  char GiocatoreDiTurno() const { return giocatore_di_turno; }
  char Pezzo(Casella c) const;
  char Pezzo(int i, int j) const;
  bool Muovi(Casella c, string dir);
private:
  Casella CasellaArrivo(Casella c, string dir) const;
  void MuoviPezzo(Casella partenza, Casella arrivo);
  bool FuoriScacchiera(Casella arrivo) const;
  bool PezzoSbagliato(Casella partenza) const;
  bool MossaNonAmmessa(Casella partenza, string dir) const;
  vector<vector<char> > scacchiera;
```

```
  char giocatore_di_turno;
};
#endif
```

Soluzione esercizio 2

La funzione più complessa dell'esercizio è `Muovi()` che riceve come parametri una casella ed una direzione ed esegue, se possibile, la mossa corrispondente. Abbiamo deciso di far restituire alla funzione un valore booleano che avverte se la mossa è stata effettuata (`true`) oppure no (`false`). In questo ultimo caso, la scacchiera rimane immutata.

Le ragioni per cui una mossa possa essere non effettuabile sono le seguenti:

- La casella di partenza o di arrivo è fuori dalla scacchiera
- La direzione non è consentita a quel pezzo (muovere una pedina in diagonale)
- La casella di arrivo è occupata da una pedina del giocatore stesso
- La casella di partenza non è occupata da una pedina del giocatore oppure dallo Zen.

Per trattare alcuni dei suddetti casi definiamo delle funzioni private specifiche, in modo da semplificare l'architettura della classe.

L'effettivo spostamento del pezzo da una casella all'altra viene anch'esso delegato ad una funzione privata, chiamata `MuoviPezzo()`.

```
// File ScacchiNibelunghi.cpp

#include "ScacchiNibelunghi.h"
#include <cassert>

ostream& operator<<(ostream& os, const Partita& p)
{
  for (int i = 0; i < DIM; i++)
    {
      for (int j = 0; j < DIM; j++)
        os << p.scacchiera[i][j];
      os << endl;
    }
  if (p.giocatore_di_turno == 'b')
    os << "Mossa al bianco";
  else if (p.giocatore_di_turno == 'n')
    os << "Mossa al nero";
  else if (p.giocatore_di_turno == 'x')
    os << "Partita finita";
  else assert(false);
  return os;
}

Partita::Partita()
  : scacchiera(DIM,vector<char>(DIM))
{
  RicominciaPartita();
}

void Partita::RicominciaPartita()
{
  int i, j;
  for (j = 0; j < DIM; j++)
    scacchiera[0][j] = 'b';
  for (i = 1; i < DIM-1; i++)
    for (j = 0; j < DIM; j++)
```

```
      scacchiera[i][j] = ' ';
  for (j = 0; j < DIM; j++)
    scacchiera[DIM-1][j] = 'n';
  scacchiera[DIM/2][DIM/2] = 'z';
  giocatore_di_turno = 'b';
}

char Partita::Pezzo(Casella c) const
{
  if (FuoriScacchiera(c))
    return ' ';
  else
    return scacchiera[c.riga][c.colonna];
}

char Partita::Pezzo(int i, int j) const
{
  return Pezzo(Casella(i,j));
}

bool Partita::Muovi(Casella c1, string dir)
{
  Casella c2 = CasellaArrivo(c1,dir);
  if (FuoriScacchiera(c1) || FuoriScacchiera(c2) ||
      MossaNonAmmessa(c1,dir) ||
      Pezzo(c1) == Pezzo(c2) || // mossa cannibale
      PezzoSbagliato(c1) ||
      (Pezzo(c1) == 'z' && Pezzo(c2) != ' ') || // zen mangia pezzo
      GiocatoreDiTurno() == 'x')  // partita finita
    return false;
  MuoviPezzo(c1,c2);
  return true;
}

Casella Partita::CasellaArrivo(Casella c, string dir) const
{
  if (dir == "nord") return Casella(c.riga-1,c.colonna);
  if (dir == "est") return Casella(c.riga,c.colonna-1);
  if (dir == "sud") return Casella(c.riga+1,c.colonna);
  if (dir == "ovest") return Casella(c.riga,c.colonna+1);
  if (dir == "nord-est") return Casella(c.riga-1,c.colonna-1);
  if (dir == "nord-ovest") return Casella(c.riga-1,c.colonna+1);
  if (dir == "sud-est") return Casella(c.riga+1,c.colonna-1);
  if (dir == "sud-ovest") return Casella(c.riga+1,c.colonna+1);
  assert(false);
  return Casella(0,0);
}

void Partita::MuoviPezzo(Casella p, Casella a)
{
  if (Pezzo(a) == 'z')
    giocatore_di_turno = 'x'; // partita finita
  else
    {
      scacchiera[a.riga][a.colonna] = scacchiera[p.riga][p.colonna];
      scacchiera[p.riga][p.colonna] = ' ';
      if (giocatore_di_turno == 'b')
        giocatore_di_turno = 'n';
      else
```

```
        giocatore_di_turno = 'b';
    }
}


bool Partita::FuoriScacchiera(Casella a) const
{
  return a.riga < 0 || a.riga >= DIM || a.colonna < 0 || a.colonna >= DIM;
}

bool Partita::PezzoSbagliato(Casella p) const
{
  return giocatore_di_turno != Pezzo(p) && Pezzo(p) != 'z';
}

bool Partita::MossaNonAmmessa(Casella p, string dir) const
{
  return Pezzo(p) != 'z' &&
    dir != "nord" &&
    dir != "sud" &&
    dir != "est" &&
    dir != "ovest";
}
```

Soluzione esercizio 3 + driver

```
// File main.cpp

#include "ScacchiNibelunghi.h"
#include <cassert>

bool EsisteMossaVincente(const Partita & p);
bool EseguiMossa(Partita& p, Casella c, string dir);

int main()
{ // non richiesto per il compito in classe
  Partita p;
  string dir;
  int i,j;

  while (p.GiocatoreDiTurno() != 'x')
    {
      cout << p << endl;
      if (EsisteMossaVincente(p))
        cout << "C'e' una mossa vincente!" << endl;
      cout << "Inserisci coordinate pezzo da muovere (";
      if (p.GiocatoreDiTurno() == 'b')
        cout << "bianco";
      else
        cout << "nero";
      cout << ") : ";
      cin >> i >> j;
      cout << "Inserisci direzione (nord/sud/est/ovest/nord-est/...) : ";
      cin >> dir;
      if (!p.Muovi(Casella(i,j),dir))
        cout << "Mossa impossibile" << endl;
    }
}
```

```
bool EsisteMossaVincente(const Partita & p)
{
  int i, j;
  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
      if (p.Pezzo(i,j) == p.GiocatoreDiTurno())
        {
          if (p.Pezzo(i+1,j) == 'z' ||
              p.Pezzo(i-1,j) == 'z' ||
              p.Pezzo(i,j-1) == 'z' ||
              p.Pezzo(i,j+1) == 'z')
            return true;
        }
  return false;
}

bool EseguiMossa(Partita& p, Casella c, string dir)
{
  bool mossa_possibile = p.Muovi(c,dir);
  assert(mossa_possibile);
  return (p.GiocatoreDiTurno() == 'x');
}
```

Soluzione compito del 25 giugno 2002

Esaminando le specifiche è facile rendersi conto che la responsabilità delle associazioni effettua e relativa deve essere attribuita alla classi Vigile e Contravvenzione rispettivamente. Conseguentemente, ciascun vigile ha come dato principale il vettore dei puntatori alle contravvenzioni che ha elevato. La contravvenzione ha invece un puntatore al veicolo a cui si riferisce.

Si noti che non c'è alcun riferimento esplicito tra il vigile a il veicolo. Infatti, coerentemente con il diagramma UML, non c'è legame diretto tra queste due classi.

Soluzione esercizio 1

```
//File vigile.h
#ifndef VIGILE_H
#define VIGILE_H

#include <vector>
#include <string>

using namespace std;

class Veicolo
{
public:
  Veicolo(string, int, string);
  string Targa() const { return targa; }
  int Potenza() const { return potenza; }
  string Tipo() const { return tipo; }
  bool operator==(const Veicolo& v) const { return targa == v.targa; }
private:
  string targa;
  int potenza;
  string tipo;
};
```

```
class Contravvenzione
{
public:
  Contravvenzione(int, string, Veicolo*);
  string Luogo() const { return luogo; }
  int Numero()const { return numero; }
  Veicolo* QualeVeicolo() const { return veicolo; }
  bool operator==(const Contravvenzione& c) const { return numero == c.numero; }
private:
  string luogo;
  int numero;
  Veicolo* veicolo;
};

class Vigile
{
  friend ostream& operator<<(ostream& os,const Vigile& vig);
public:
  Vigile(string,string);
  void EffettuaContravvenzione(Contravvenzione*);
  void EliminaContravvenzione(Contravvenzione*);
  string Nome() const {return nome;}
  string Matricola()const { return matricola; }
  Contravvenzione* ContravvenzioneEffettuata(int) const;
  int NumeroContravvenzioni() const { return contravvenzioni.size(); }
private:
  string nome;
  string matricola;
  vector <Contravvenzione*> contravvenzioni;
  int Cerca(Contravvenzione*) const;
};
#endif
```

Soluzione esercizio 2

```
//File vigile.cpp

#include "vigile.h"
#include <iostream>
#include <cassert>

Vigile::Vigile(string n,string m)
{
  nome = n;
  matricola = m;
}

void Vigile::EffettuaContravvenzione(Contravvenzione* c)
{
  contravvenzioni.push_back(c);
}

void Vigile::EliminaContravvenzione(Contravvenzione* c)
{
  int i = Cerca(c);
  assert(i != -1);
  contravvenzioni.erase(contravvenzioni.begin()+i);
}
```

```
Contravvenzione* Vigile::ContravvenzioneEffettuata(int i) const
{
  assert(i >= 0 && i < contravvenzioni.size());
  return contravvenzioni[i];
}

int Vigile::Cerca(Contravvenzione* c) const
{
  for(int i = 0; i < contravvenzioni.size(); i++)
    if(*(contravvenzioni[i]) == *c)
      return i;
  return -1;
}

Contravvenzione::Contravvenzione(int n, string l, Veicolo* v)
{
  numero = n;
  luogo = l;
  veicolo = v;
}

Veicolo::Veicolo(string n, int p, string t)
{
  targa = n;
  potenza = p;
  tipo = t;
}

ostream& operator<<(ostream& os, const Vigile& vig)
{
  os << " VIGILE:  " << vig.Nome()
     << "                  MATRICOLA:  " << vig.Matricola()<<endl;
  os << " ================================================"
     <<  "============================ "<<endl;
  if(vig.contravvenzioni.size()==0)
    os << " Non ha effettuato contravvenzioni"<< endl;

  else
  {
    os << " Le sue contravvenzioni sono:"<< endl;
    for (int i = 0; i < vig.contravvenzioni.size(); i++)
    {
      os << " -- NUMERO:   " << vig.contravvenzioni[i]->Numero()
         << "     LUOGO:  " << vig.contravvenzioni[i]->Luogo()
         << "  VEICOLO: " << vig.contravvenzioni[i]->QualeVeicolo()->Tipo()
         << endl;
       os <<"   TARGA:  "<< vig.contravvenzioni[i]->QualeVeicolo()
            ->Targa()<<'\t'<<"  POTENZA:  "
          << vig.contravvenzioni[i]->QualeVeicolo()->Potenza()
          << endl;

     }
  }
  os << " ================================================"
     <<  "============================ "<<endl;
  return os;
}
```

Si noti come nella funzione `RipetiVeicolo()` qui sotto il confronto tra veicoli venga fatto a livello

di oggetti e non di puntatori. Infatti l'uguaglianza tra i puntatori è una condizione sufficiente ma non necessaria affinché due oggetti siano uguali. A questo scopo torna utile la definizione dell'operatore == per la classe `Veicolo`.

Soluzione esercizio 3 e 4 + driver

```
// File main.cpp
#include "vigile.h"
#include <iostream>

bool ComparaMotoAuto(const Vigile& v);
bool RipetiVeicolo(const Vigile& v);
void StampaStato(const Vigile& v);

int main()
{
  Vigile vig("Bartolomeo Pestalozzi","6969");
  Veicolo a1("Ferrari",30000,"Auto");
  Veicolo a2("Seat",1200,"Auto");
  Veicolo a3("Fiat",1100,"Auto");
  Veicolo m4("Ciao",900,"Moto");
  Veicolo m3("Suzuky",800,"Moto");
  Veicolo m2("Gilera",700,"Moto");
  Veicolo m1("Honda",150,"Moto");
  Contravvenzione c1(7,"Milano",&a1);
  Contravvenzione c2(06,"Torino",&m2);
  Contravvenzione c3(3,"Bologna",&m3);
  Contravvenzione c4(1,"Marte",&m4);
  Contravvenzione c5(2,"Saturno",&a3);
  Contravvenzione c6(5,"Giove",&a2);
  Contravvenzione c7(9,"Venere",&m1);
  Contravvenzione c8(4,"Roma",&a1);

  vig.EffettuaContravvenzione(&c1);
  vig.EffettuaContravvenzione(&c2);
  vig.EffettuaContravvenzione(&c3);
  vig.EffettuaContravvenzione(&c4);
  vig.EffettuaContravvenzione(&c5);
  vig.EffettuaContravvenzione(&c6);
  vig.EffettuaContravvenzione(&c7);
  StampaStato(vig);
  vig.EffettuaContravvenzione(&c8);
  StampaStato(vig);
  vig.EliminaContravvenzione(&c2);
  vig.EliminaContravvenzione(&c3);
  vig.EliminaContravvenzione(&c4);
  vig.EliminaContravvenzione(&c5);
  StampaStato(vig);
  vig.EliminaContravvenzione(&c1);
  vig.EliminaContravvenzione(&c8);
  vig.EliminaContravvenzione(&c6);
  vig.EliminaContravvenzione(&c7);
  StampaStato(vig);
}

bool RipetiVeicolo(const Vigile& v)
{
  for(int i = 0; i < v.NumeroContravvenzioni(); i++)
    for(int j = i + 1; j < v.NumeroContravvenzioni(); j++)
```

```
      if(*(v.ContravvenzioneEffettuata(i)->QualeVeicolo()) ==
         *(v.ContravvenzioneEffettuata(j)->QualeVeicolo()))
        return true;
   return false;
}

bool  ComparaMotoAuto(const Vigile& v)
{
  int conta_moto = 0, conta_auto = 0;
  for(int i = 0; i < v.NumeroContravvenzioni(); i++)
  {
    if(v.ContravvenzioneEffettuata(i)->QualeVeicolo()->Tipo() == "Moto")
      conta_moto++;
    else
      conta_auto++;
  }
  return conta_moto > conta_auto;
}

void StampaStato(const Vigile& v)
{
  cout << v;
  if (RipetiVeicolo(v))
    cout << v.Nome() << " ha dato piu' di una multa a uno stesso veicolo" << endl;
  else
    cout << v.Nome() << " non ha dato piu' di una multa a uno stesso veicolo" << endl;

  if(ComparaMotoAuto(v))
    cout << v.Nome() << " ha dato piu' multe alle moto"<< endl;
  else
    cout << v.Nome() << " le multe alle moto non sono maggiori di quelle delle auto" <<endl;
  cout << endl << "Premi invio per continuare ";
  cin.get();
}
```

Soluzione compito del 15 luglio 2002

Per rappresentare i dati del sistema di prenotazione utilizziamo un vettore di prenotazioni. Ciascuna prenotazione è rappresentata da un oggetto della classe `Prenotazione` che contiene un puntatore alla persona, il settore richiesto, la flessibilità e lo stato della prenotazione. Lo stato è codificato nel seguente modo:

1. prenotazione andata a buon fine
2. prenotazione accettata, ma con cambio di settore
3. prenotazione in attesa

Oltre al vettore, la classe `Teatro` ha come dati quattro costanti (la capienza di ciascun settore ed il suo prezzo) più due variabili che rappresentano il numero di posti correntemente occupati nei due settori.

Soluzione esercizio 1

```
// file teatro.h
#ifndef TEATRO_H
#define TEATRO_H
#include "persona.h"
#include <iostream>
```

```
#include <vector>

using namespace std;

class Prenotazione
{
public:
  Prenotazione(Persona *p, bool pl, bool f)
  { prenotato = p; richiesta_platea = pl; flessibile = f; }
  Persona *prenotato;
  bool richiesta_platea;
  bool flessibile;
  unsigned status; // 1: soddisfatta, 2: soddisfatta con cambio, 3: in attesa
};

class Teatro
{
  friend ostream& operator<<(ostream & os, const Teatro& t);
public:
  Teatro(int pp, int pg, int cp, int cg);
  int InserisciPrenotazione(Persona* p, bool t, bool c);
  void EliminaPrenotazione(Persona* p);
  int StatoPrenotazione(Persona* p) const;
  int NumPrenotazioni() const { return prenotazioni.size(); }
  int PostiPlatea() const { return posti_platea; }
  int PostiGalleria() const { return posti_galleria; }
  int PrezzoPlatea() const { return prezzo_platea; }
  int PrezzoGalleria() const { return prezzo_galleria; }
  int PrenotatiPlatea() const { return prenotati_platea; }
  int PrenotatiGalleria() const { return prenotati_galleria; }
private:
  vector<Prenotazione> prenotazioni;
  int posti_platea;
  int posti_galleria;
  int prezzo_platea;
  int prezzo_galleria;
  int prenotati_platea;
  int prenotati_galleria;
  int CercaPersona(Persona* p) const;
  void RecuperaPrenotazione(int i, string posto);
};
#endif
```

Soluzione esercizio 2

La funzione `InserisciPrenotazione()` crea una nuova prenotazione e ne determina lo stato in base alla disponibilità del teatro. La prenotazione viene inserita in ultima posizione nel vettore delle prenotazioni. L'inserimento avviene in ultima posizione perché cosìì il vettore si mantiene ordinato "cronologicamente". Il numero di posti correntemente occupati viene ovviamente anch'esso modificato in base allo disponibilità. Ad esempio se una persona richiede la platea ed è flessibile, e la platea è piena mentre sono disponibili posti in galleria, viene assegnata alla prenotazione lo stato 2 (soddisfatta con cambio) e viene occupato un posto in galleria.

La funzione `EliminaPrenotazione()` elimina la prenotazione della persona passata come parametro. Prima di eliminare fisicamente la prenotazione, vengono aggiornati i contatori dei posti occupati e viene memorizzato nella variabile `posto_recuperato` il settore del posto eventualmente lasciato libero dalla persona che cancella la prenotazione.

Se viene lasciato libero un posto, una delle prenotazioni successive può essere soddisfatta (o cambiata di settore). Questa operazione viene eseguita da una funzione privata specifica, chiamata `RecuperaPrenotazione()`, che cerca nel vettore una prenotazione che ha bisogno di un posto nel settore lasciato libero e in caso

ne cambia lo stato. Come evidenziato nel testo, è anche possibile che una seconda prenotazione possa essere modificata nel caso la prima si riferisca ad un cambio di settore. In questo caso, la funzione `RecuperaPrenotazione()` si invoca ricorsivamente per eseguire il secondo cambio.

```
// teatro.cpp
#include "teatro.h"
#include <cassert>

Teatro::Teatro(int pp, int pg, int cp, int cg)
{
  posti_platea = pp;
  posti_galleria = pg;
  prezzo_platea = cp;
  prezzo_galleria = cg;
  prenotati_platea = 0;
  prenotati_galleria = 0;
}

int Teatro::InserisciPrenotazione(Persona* p, bool t, bool c)
{
  if (CercaPersona(p) != -1)
    return 0;
  Prenotazione pr(p,t,c);
  if (t)
    {
      if (prenotati_platea < posti_platea)
        {
          pr.status = 1;
          prenotati_platea++;
        }
      else if (c && prenotati_galleria < posti_galleria)
        {
          pr.status = 2;
          prenotati_galleria++;
        }
      else
        pr.status = 3;
    }
  else
    {
      if (prenotati_galleria < posti_galleria)
        {
          pr.status = 1;
          prenotati_galleria++;
        }
      else if (c && prenotati_platea < posti_platea)
        {
          pr.status = 2;
          prenotati_platea++;
        }
      else
        pr.status = 3;
    }
  prenotazioni.push_back(pr);
  return pr.status;
}

void Teatro::EliminaPrenotazione(Persona* p)
{
  int i;
```

```
  string posto_recuperato;

  i = CercaPersona(p);
  assert (i != -1);
  if (prenotazioni[i].richiesta_platea && prenotazioni[i].status == 1
      || !prenotazioni[i].richiesta_platea && prenotazioni[i].status == 2)
    {
      posto_recuperato = "platea";
      prenotati_platea--;
    }
  else if (!prenotazioni[i].richiesta_platea && prenotazioni[i].status == 1
      || prenotazioni[i].richiesta_platea && prenotazioni[i].status == 2)
    {
      posto_recuperato = "galleria";
      prenotati_galleria--;
    }
  else
    posto_recuperato = "no";
  prenotazioni.erase(prenotazioni.begin() + i);
  if (posto_recuperato != "no")
    RecuperaPrenotazione(i,posto_recuperato);
}

void Teatro::RecuperaPrenotazione(int n, string posto)
{
  string recupera_un_altro_posto = "no";
  int j;

  if (posto == "platea")
    // la persona eliminanda occupava un posto in platea,
    // si recupera una prenotazione in platea
    {
      for (j = n; j < prenotazioni.size(); j++)
        {
          if (prenotazioni[j].status == 2
           && prenotazioni[j].richiesta_platea
           && prenotazioni[j].flessibile)
            {
              prenotazioni[j].status = 1;
              prenotati_galleria--;
              prenotati_platea++;
              recupera_un_altro_posto = "galleria";
              break;
            }
          else if (prenotazioni[j].status == 3
                && prenotazioni[j].richiesta_platea)
            {
              prenotazioni[j].status = 1;
              prenotati_platea++;
              break;
            }
          else if (prenotazioni[j].status == 3
                && !prenotazioni[j].richiesta_platea
                &&  prenotazioni[j].flessibile)
            {
              prenotazioni[j].status = 2;
              prenotati_platea++;
              break;
            }
```

```
        }
    }
  else
    // la persona eliminanda occupava un posto in galleria,
    // si recupera una prenotazione in galleria
    {
      for (j = n; j < prenotazioni.size(); j++)
        {
          if (prenotazioni[j].status == 2
           && !prenotazioni[j].richiesta_platea
           && prenotazioni[j].flessibile)
            {
              prenotazioni[j].status = 1;
              prenotati_platea--;
              prenotati_galleria++;
              recupera_un_altro_posto = "platea";
              break;
            }
          else if (prenotazioni[j].status == 3
                && !prenotazioni[j].richiesta_platea)
            {
              prenotazioni[j].status = 1;
              prenotati_galleria++;
              break;
            }
          else if (prenotazioni[j].status == 3
                && prenotazioni[j].richiesta_platea
                && prenotazioni[j].flessibile)
            {
              prenotazioni[j].status = 2;
              prenotati_galleria++;
              break;
            }
        }
    }
  if (recupera_un_altro_posto != "no")
    RecuperaPrenotazione(j+1, recupera_un_altro_posto);
}

int Teatro::StatoPrenotazione(Persona* p) const
{
  int i = CercaPersona(p);
  if (i == -1) return 0;
  else return prenotazioni[i].status;
}

int Teatro::CercaPersona(Persona* p) const
{
  for (unsigned i = 0; i < prenotazioni.size(); i++)
    if (*p == *(prenotazioni[i].prenotato))
      return i;
  return -1;
}

ostream& operator<<(ostream & os, const Teatro& t)
{
  os << endl << "---- Stato teatro ----" << endl;
  os << "Numero posti in platea : " << t.posti_platea << endl;
  os << "Numero posti in galleria : " << t.posti_galleria << endl;
```

```
  os << "Prezzo posti in platea : " << t.prezzo_platea << endl;
  os << "Prezzo posti in galleria : " << t.prezzo_galleria << endl;
  os << "Posti prenotati in platea : " << t.prenotati_platea << endl;
  os << "Posti prenotati in galleria : " << t.prenotati_galleria << endl;
  for (int i = 0; i < t.prenotazioni.size(); i++)
    {
      os << i+1 << " : " << t.prenotazioni[i].prenotato->Nome() << " richiesta ";
      if (t.prenotazioni[i].richiesta_platea)
        os << "platea ";
      else
        os << "galleria ";
      if (t.prenotazioni[i].flessibile)
        os << "flessibile ";
      os << " - status : ";
      if (t.prenotazioni[i].status == 1)
        os << " ok ";
      else if (t.prenotazioni[i].status == 2)
        os << " cambio settore ";
      else if (t.prenotazioni[i].status == 3)
        os << "in attesa";
      os << endl;
    }
  os << endl;
  return os;
}
```

Soluzione esercizi3 e 4 + driver

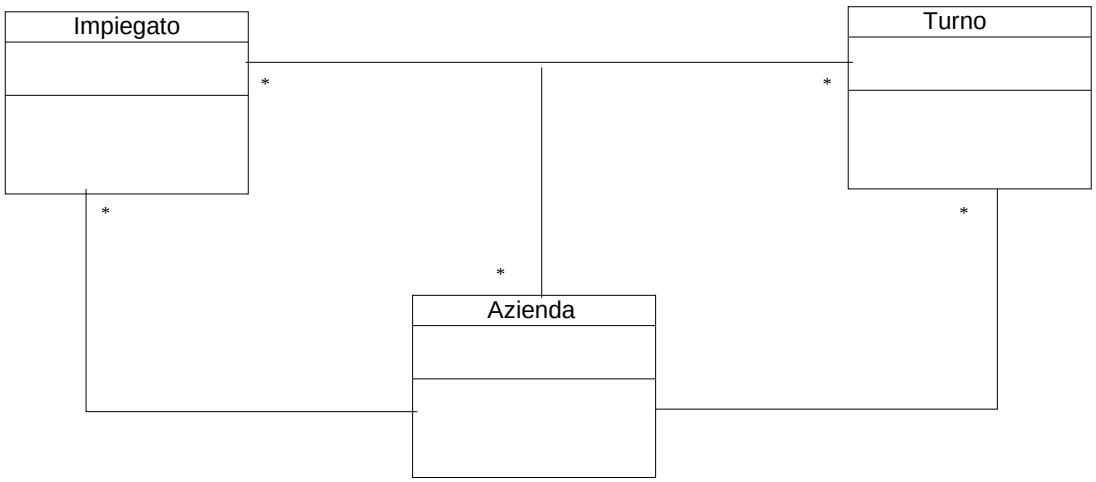
```
// main.cpp
#include "teatro.h"

int IncassoTeatro(const Teatro& t);
void InserisciPrenotazioni(Teatro& t, vector<Prenotazione> v);

int main()
{
  int pp, pg, cp, cg, scelta, s;
  string nome;
  char ch1,ch2;
  bool platea, flessibile;
  Persona* p;

  cout << "Gestione prenotazioni teatro" << endl;
  cout << "Numero posti in platea : ";
  cin >> pp;
  cout << "Numero posti in galleria : ";
  cin >> pg;
  cout << "Costo posti in platea : ";
  cin >> cp;
  cout << "Costo posti in galleria : ";
  cin >> cg;
  Teatro t(pp,pg,cp,cg);

  do
    {
      cout << t;
      cout << "Quale operazione vuoi effettuare?\n"
           << "1: Inserimento prenotazione\n"
           << "2: Eliminazione prenotazione\n"
```

```
     << "3: Calcola incasso\n"
     << "4: Inserisci vettore prenotazioni\n"
     << "0: Esci\n\n"
     << "Scelta:  ";
cin >> scelta;
cout << endl;
switch (scelta)
  {
  case 1:
    cout << "Nome : ";
    cin >> nome;
    p = new Persona(nome);
    cout << "Platea o galleria? (p/g) : ";
    cin >> ch1;
    if (ch1 == 'p')
      platea = true;
    else
      platea = false;
    cout << "Accetta cambi di settore (s/n) : ";
    cin >> ch2;
    if (ch2 == 's')
      flessibile = true;
    else
      flessibile = false;
    s = t.InserisciPrenotazione(p,platea,flessibile);
    cout << endl;
    switch (s)
      {
      case 0:
        cout << "Persona gia' prenotata" << endl;
        break;
      case 1:
        cout << "Ok, prenotazione effettuata" << endl;
        break;
      case 2:
        cout << "Prenotazione effettuata con cambio" << endl;
        break;
      case 3:
        cout << "Prenotazione in attesa" << endl;
      }
    break;
  case 2:
    cout << "Nome : ";
    cin >> nome;
    p = new Persona(nome);
    t.EliminaPrenotazione(p);
    break;
  case 3:
    cout << "L'incasso e' : " << IncassoTeatro(t) << endl;
    break;
  case 4:
    {
      vector<Prenotazione> v(5,Prenotazione(NULL,true,true));
      v[0] = Prenotazione(new Persona("p1"), true, true);
      v[1] = Prenotazione(new Persona("p2"), true, false);
      v[2] = Prenotazione(new Persona("p3"), false, true);
      v[3] = Prenotazione(new Persona("p4"), false, false);
      v[4] = Prenotazione(new Persona("p1"), true, false);
      InserisciPrenotazioni(t,v);
```

```
            break;
          }
        }
     }
  while (scelta != 0);
}

int IncassoTeatro(const Teatro& t)
{
  return t.PrezzoPlatea() * t.PrenotatiPlatea() +
    t.PrezzoGalleria() * t.PrenotatiGalleria();
}

void InserisciPrenotazioni(Teatro& t, vector<Prenotazione> v)
{
  for (int i = 0; i < v.size(); i++)
    {
      if (t.StatoPrenotazione(v[i].prenotato) != 0)
        t.EliminaPrenotazione(v[i].prenotato);
     t.InserisciPrenotazione(v[i].prenotato,v[i].richiesta_platea,v[i].flessibile);
    }
}
```

Soluzione compito del 9 settembre 2002

Anche se non richiesto, per chiarezza mostriamo il diagramma delle classi UML (incompleto).



Si noti che abbiamo deciso di modellare la relazione tra impiegati e turni come una associazione a tre (Azienda, Impiegato e Turno). La responsabilità di questa associazione, ed anche delle altre associazioni ricade sulla classe Azienda.

Per memorizzare gli impiegati assunti ed i turni di lavoro vengono utilizzati come di consueto due vettori di puntatori. Per memorizzare la assegnazioni degli impiegati ai turni si è scelto di utilizzare un terzo vettore, formato da coppie impiegato/turno. Per memorizzare ciascuna coppia si è utilizzata una classe che contiene due interi, che corrispondono agli indici dell'impiegato e del turno nei vettori corrispondenti dell'oggetto della classe `Azienda`. La memorizzazione di puntatori invece che di indici sarebbe stata ugualmente possibile, ma leggermente più complessa da gestire.

Una soluzione alternativa sarebbe stata quella di considerare l'associazione a tre come associazione solo tra Inpiegato e Turno e dare la responsabilità di questa associazione alla classe Impiegato. In questo caso, la avremmo dovuto definire una nuova classe per gestire un impiegato con incluso anche il vettore dei suoi turni di lavoro. In questo casi la classe `Azienda` avrebbe avuto come dati solo i due vettori contenenti i turni disponibili e gli impiegati. Questa soluzione, prescelta dalla maggioranza degli studenti all'esame, viene lasciata per esercizio.

Soluzione esercizio 1

```
// file Azienda.h
#ifndef AZIENDA_H
#define AZIENDA_H
#include <string>
#include <vector>
#include "impiegato.h"
#include "turno.h"

class Assegnazione
{// un oggetto rappresenta l'assegnazione di una persona ad un turno
public:
  Assegnazione(unsigned i, unsigned t) { imp = i; turno = t; }
  unsigned imp, turno;
};

class Azienda
{
  friend ostream& operator<<(ostream &, const Azienda&);
 public:
  Azienda(unsigned max_imp);
  unsigned Impiegati() const { return impiegati.size(); }
  unsigned Turni() const { return turni.size(); }
  unsigned Assegnazioni() const { return assegnazioni.size(); }
  void Assumi(Impiegato* i);
  void CreaTurno(Turno* t);
  void AssegnaTurno(Impiegato* i, Turno* t);
  Impiegato* VediImpiegato(unsigned i) const;
  Turno* VediTurno(unsigned i) const;
  pair<Impiegato*,Turno*> VediAssegnazione(unsigned i) const;
 private:
  unsigned max_impiegati;
  vector<Impiegato*> impiegati;
  vector<Turno*> turni;
  vector<Assegnazione> assegnazioni;
  int CercaImpiegato(Impiegato* i) const;
  int CercaTurno(Turno* t) const;
};
#endif
```

Soluzione esercizio 2

```
// file azienda.cpp
#include "azienda.h"
#include <iostream>
#include <cassert>

Azienda::Azienda(unsigned max_imp)
{
  max_impiegati = max_imp;
}

void Azienda::Assumi(Impiegato* i)
{
  assert(impiegati.size() < max_impiegati);
  if (CercaImpiegato(i) == -1)
    impiegati.push_back(i);
}
```

```
void Azienda::CreaTurno(Turno* t)
{
  assert(CercaTurno(t) == -1);
  turni.push_back(t);
}

void Azienda::AssegnaTurno(Impiegato* i, Turno* t)
{
  int j = CercaImpiegato(i);
  int k = CercaTurno(t);
  assert(j != -1 && k != -1);
  for (unsigned h = 0; h < assegnazioni.size(); h++)
    if (assegnazioni[h].imp == unsigned(j))
      assert(Disgiunti(turni[assegnazioni[h].turno], t));
  assegnazioni.push_back(Assegnazione(j,k));
}

Impiegato* Azienda::VediImpiegato(unsigned i) const
{
  assert (i < impiegati.size());
  return impiegati[i];
}

Turno* Azienda::VediTurno(unsigned i) const
{
  assert (i < turni.size());
  return turni[i];
}

pair<Impiegato*, Turno*> Azienda::VediAssegnazione(unsigned i) const
{

  assert (i < assegnazioni.size());
  return make_pair(impiegati[assegnazioni[i].imp],
                   turni[assegnazioni[i].turno]);
}

int Azienda::CercaImpiegato(Impiegato* i) const
{
  for (unsigned j = 0; j < impiegati.size(); j++)
    if (*(impiegati[j]) == *i)
      return j;
  return -1;
}

int Azienda::CercaTurno(Turno* t) const
{
  for (unsigned j = 0; j < turni.size(); j++)
    if (turni[j]->Start() == t->Start() && turni[j]->Stop() == t->Stop())
      return j;
  return -1;
}

ostream& operator<<(ostream & os, const Azienda& az)
{// non richiesto per il compito in classe
  unsigned i;

  cout << "Stampa azienda (max " << az.max_impiegati << ") dipendenti" << endl;
  cout << "Impiegati:" << endl;
```

```
  for (i = 0; i < az.impiegati.size(); i++)
    cout << i+1 << " : " << az.impiegati[i]->Nome() << endl;
  cout << "Turni:" << endl;
  for (i = 0; i < az.turni.size(); i++)
    cout << i+1 << " : " << az.turni[i]->Start() << "-" << az.turni[i]->Stop() << endl;
  cout << "Assegnazioni:" << endl;
  for (i = 0; i < az.assegnazioni.size(); i++)
    cout << i+1 << " : " << az.impiegati[az.assegnazioni[i].imp]->Nome() << " ==> "
         << az.turni[az.assegnazioni[i].turno]->Start() << "-"
         << az.turni[az.assegnazioni[i].turno]->Stop() << endl;
  return os;
}
```

Si noti la diversa gestione del caso di tentativo di inserire un dato già presente nel caso degli impiegati e nel caso dei turni. Nel primo caso, infatti, le specifiche dicono esplicitamente che bisogna considerare tali possibilità e che il modo di rispondere è semplicemente quello di non compiere alcuna azione. Nel caso dei turni, questa eventualità è esclusa in base alle precondizioni, e il modo scelto di gestire le precondizioni è di invocare la funzione `assert()`. Si noti che nel primo caso si risponde ad una esigenza di *correttezza* mentre nel secondo ad una esigenza di *robustezza*.

Soluzione esercizi 3 e 4 + driver

```
// File main.cpp
#include "azienda.h"
#include <iostream>

bool PresenzaTurno(const Azienda& az, unsigned start, unsigned stop);
unsigned TotaleOreLavoro(const Azienda& az);

int main()
{
  int dip, scelta, start, stop;
  string nome;
  Impiegato* p;
  Turno *t;

  cout << "Gestione turni aziendali" << endl;
  cout << "Numero massimo dipendenti : ";
  cin >> dip;

  Azienda az(dip);
  do
    {
      cout << az;
      cout << "Quale operazione vuoi effettuare?\n"
           << "1: Inserimento impiegato\n"
           << "2: Inserimento turno\n"
           << "3: Inserimento assegnazione\n"
           << "4: Verifica presenza turno\n"
           << "5: Conta ore lavorate\n"
           << "0: Esci\n\n"
           << "Scelta:  ";
      cin >> scelta;
      cout << endl;
      switch (scelta)
        {
        case 1:
          cout << "Nome : ";
          cin >> nome;
          p = new Impiegato(nome);
```

```
          az.Assumi(p);
          break;
        case 2:
          cout << "Inizio turno : ";
          cin >> start;
          cout << "Fine turno : ";
          cin >> stop;
          t = new Turno(start,stop);
          az.CreaTurno(t);
          break;
        case 3:
          cout <<  "Nome : ";
          cin >> nome;
          p = new Impiegato(nome);
          cout << "Inizio turno : ";
          cin >> start;
          cout << "Fine turno : ";
          cin >> stop;
          t = new Turno(start,stop);
          az.AssegnaTurno(p,t);
          break;
        case 4:
          cout << "Inizio turno : ";
          cin >> start;
          cout << "Fine turno : ";
          cin >> stop;
          if (PresenzaTurno(az,start,stop))
            cout << "Il turno esiste" << endl;
          else
            cout << "Il turno non esiste" << endl;
          break;
        case 5:
          cout << "Ore totali lavorate : " << TotaleOreLavoro(az) << endl;
          break;
        }
    }
  while (scelta != 0);
}

bool PresenzaTurno(const Azienda& az, unsigned start, unsigned stop)
{
  for (unsigned i = 0; i < az.Turni(); i++)
    {
      if (az.VediTurno(i)->Start() == start
          && az.VediTurno(i)->Stop() == stop)
        return true;
    }
  return false;
}

unsigned TotaleOreLavoro(const Azienda& az)
{
  unsigned conta = 0;
  for (unsigned i = 0; i < az.Assegnazioni(); i++)
    conta += az.VediAssegnazione(i).second->NumOre();
  return conta;
}
```

Classi e funzioni predefinite

```
// file Impiegato.h
#ifndef IMPIEGATO_H
#define IMPIEGATO_H
#include <string>

using namespace std;

class Impiegato
{
 public:
  Impiegato(string n) { nome = n; }
  string Nome() const { return nome; }
  bool operator==(const Impiegato& p)
    const { return nome == p.nome; }
 private:
  string nome;
};

#endif
```

Soluzione compito del 25 settembre 2002

La classe `Disco` conterrà come dati un vettore di puntatori a `File`, una stringa per il nome e due reali che memorizzano rispettivamente la dimensione totale e lo spazio libero. Si noti che lo spazio libero è ridondante in quanto questo può essere calcolato dalla somma delle dimensione degli elementi del vettore. Si è deciso di memorizzarlo esplicitamente per rendere più efficiente l'accesso a questo dato.

Soluzione esercizio 1: file `Disco.h`

```
// File Disco.h
#ifndef DISCO_H
#define DISCO_H
#include <iostream>
#include <vector>
#include "File.h"

class Disco
{ friend ostream& operator<<(ostream& os, Disco& m);
 public:
  Disco(float dim, string nome);
  string Nome() const { return nome; }
  bool operator==(const Disco& d) const { return nome == d.nome; }
  float DimensioneTotale() const { return dimensione_totale; }
  float SpazioLibero() const { return spazio_libero; }
  void ScriviFile(File*);
  void EliminaFile(File*);
  void CancellaTutto();
 private:
  int CercaFile(File* f) const;
  float dimensione_totale, spazio_libero;
  vector<File*> dati;
  string nome;
};
#endif
```

I dati della classe `Gestore` consistono in un vettore di dischi ed un vettore di file. Non è necessario prevedere un vettore per le associazione tra dischi e file, perché queste sono gestite direttamente dalla classe `Disco` (come da diagramma UML).

Soluzione esercizio 1: file `Gestore.h`

```
// File Gestore.h
#ifndef GESTORE_H
#define GESTORE_H
#include <iostream>
#include <vector>
#include "File.h"
#include "Disco.h"

class Gestore
{ friend ostream& operator<<(ostream& os, Gestore& m);
 public:
  unsigned NumFile() const { return file.size(); }
  unsigned NumDischi() const { return dischi.size(); }
  void AggiungiFile(File* f);
  void AggiungiDisco(Disco* d);
  void ScriviFileSuDisco(File* f, Disco* d);
  bool VerificaAssegnazione(const vector<unsigned>& v);
  void EseguiAssegnazione(const vector<unsigned>& v);
  void PulisciDischi();
  Disco* VediDisco(unsigned i) const { return dischi[i]; }
  File* VediFile(unsigned i) const { return file[i]; }
 private:
  int CercaDisco(Disco* d) const;
  int CercaFile(File* f) const;
  vector<File*> file;
  vector<Disco*> dischi;
};
#endif
```

Soluzione esercizio 2: file `Disco.cpp`

```
// File Disco.cpp
#include "Disco.h"
#include <cassert>

Disco::Disco(float dim, string n)
{
  dimensione_totale = dim;
  spazio_libero = dim;
  nome = n;
}

void Disco::ScriviFile(File* f)
{
  if (CercaFile(f) == -1)
    {
      dati.push_back(f);
      spazio_libero -= f->Dimensione();
      assert(spazio_libero >= 0);
    }
}

void Disco::EliminaFile(File* f)
```

```
{
  int i = CercaFile(f);
  if (i == -1)
    {
      dati.erase(dati.begin() + i);
      spazio_libero += f->Dimensione();
    }
}

void Disco::CancellaTutto()
{
  dati.clear();
  spazio_libero = dimensione_totale;
}

int Disco::CercaFile(File* f) const
{
  for (unsigned i = 0; i < dati.size(); i++)
    if (*f == *(dati[i]))
      return i;
  return -1;
}

ostream& operator<<(ostream& os, Disco& d)
{
  os << "Dimensione Totale = " << d.dimensione_totale << ", Spazio Libero = "
     << d.spazio_libero << " [";
  for (unsigned i = 0; i < d.dati.size(); i++)
    os << *(d.dati[i]) << " ";
  os << "]";
  return os;
}
```

Soluzione esercizi 2 e 4: file `Gestore.cpp`

Si noti che l'esercizio 4 è stato risolto dotando tutte le classe interessate dell'operatore `<<`, in modo tale che l'operatore `<<` della classe `Gestore` possa sfruttare gli operatori omologhi delle classe `File` e `Disco`.

```
// File Gestore.cpp
#include "Gestore.h"
#include <cassert>

void Gestore::AggiungiFile(File* f)
{
  file.push_back(f);
}

void Gestore::AggiungiDisco(Disco* d)
{
  dischi.push_back(d);
}

void Gestore::ScriviFileSuDisco(File* f, Disco* d)
{
  int i,j;
  i = CercaFile(f);
  j = CercaDisco(d);
  assert(i != -1 && j != -1);
  dischi[j]->ScriviFile(f);
}
```

```
bool Gestore::VerificaAssegnazione(const vector<unsigned>& v)
{
  // esistono due strategie per scrivere questa funzione:
  // 1. scrivere effettivamente sui dischi e poi cancellarli
  // 2. simulare la scrittura che un vettore ausiliario che
  //      rappresenti le dimensioni dei dischi
  // si e' preferita la soluzione 2 perche' di uso piu' generale
  // (ad esempio nel caso non sia possibile cancellare i dischi)

  unsigned i;

  assert (v.size() == file.size());
  for (i = 0; i < dischi.size(); i++)
    assert (dischi[i]->DimensioneTotale() == dischi[i]->SpazioLibero());

  vector<float> spazio_dischi(dischi.size());
  for (i = 0; i < spazio_dischi.size(); i++)
    spazio_dischi[i] = dischi[i]->DimensioneTotale();
  for (i = 0; i < v.size(); i++)
    spazio_dischi[v[i]] -= file[i]->Dimensione();
  for (i = 0; i < spazio_dischi.size(); i++)
    if (spazio_dischi[i] < 0)
      return false;
  return true;
}

void Gestore::EseguiAssegnazione(const vector<unsigned>& v)
{
  assert (v.size() == file.size());
  PulisciDischi();
  assert(VerificaAssegnazione(v));
  for (unsigned i = 0; i < v.size(); i++)
    dischi[v[i]]->ScriviFile(file[i]);
}

void Gestore::PulisciDischi()
{
  for (unsigned i = 0; i < dischi.size(); i++)
    dischi[i]->CancellaTutto();
}

int Gestore::CercaDisco(Disco* d) const
{
  for (unsigned i = 0; i < dischi.size(); i++)
    if (*d == *(dischi[i]))
      return i;
  return -1;
}

int Gestore::CercaFile(File* f) const
{
  for (unsigned i = 0; i < file.size(); i++)
    if (*f == *(file[i]))
      return i;
  return -1;
}

ostream& operator<<(ostream& os, Gestore& g)
```

```
{
  unsigned i;
  os << "Stato del gestore dei dischi" << endl;
  os << "Dischi: " << endl;
  for (i = 0; i < g.NumDischi(); i++)
    os << i << " : " << *(g.dischi[i]) << endl;
  os << "File: " << endl;
  for (i = 0; i < g.NumFile(); i++)
    os << i << " : " << *(g.file[i])<< endl;
  return os;
}
```

Soluzione esercizi 3 + driver

```
// File Gestore.cpp
#include "Gestore.h"

Gestore* AssegnaFile(Disco* d1, Disco* d2, vector<File*> v);

int main()
{
  Gestore g;
  Disco *d, *d2;
  File* f;
  int scelta;
  unsigned i,j, n;
  string nome;
  float dim;

  cout << "Gestione file e dischi" << endl;
  do
    {
      cout << g;
      cout << "Quale operazione vuoi effettuare?\n"
           << "1: Inserimento disco\n"
           << "2: Inserimento file\n"
           << "3: Scrittura di file su disco\n"
           << "4: Esegui Assegnazione\n"
           << "5: Verifica Assegnazione\n"
           << "6: Cancella tutti\n"
           << "7: Assegnazione file a due dischi (funzione esterna)\n"
           << "0: Esci\n\n"
           << "Scelta:  ";
      cin >> scelta;
      cout << endl;
      switch (scelta)
        {
        case 1:
          cout << "Nome disco: ";
          cin >> nome;
          cout << "Capacita' disco : ";
          cin >> dim;
          d = new Disco(dim,nome);
          g.AggiungiDisco(d);
          break;
        case 2:
          cout << "Nome file: ";
          cin >> nome;
          cout << "Dimensione file : ";
```

```
  cin >> dim;
  f = new File(nome,dim);
  g.AggiungiFile(f);
  break;
case 3:
  cout << "Indice disco: ";
  cin >> i;
  cout << "Indice file: ";
  cin >> j;
  g.ScriviFileSuDisco(g.VediFile(j),g.VediDisco(i));
  break;
case 4:
  {
    vector<unsigned> v(g.NumFile());
    for (i = 0; i < v.size(); i++)
      {
        cout << "Scrivi il file di indice " << i << " sul disco : ";
        cin >> j;
        v[i] = j;
      }
    g.EseguiAssegnazione(v);
    break;
  }
case 5:
  {
    vector<unsigned> v(g.NumFile());
    for (i = 0; i < v.size(); i++)
      {
        cout << "Il file di indice " << i << " andrebbe sul disco : ";
        cin >> v[i];
      }
    if (g.VerificaAssegnazione(v))
      cout << "Assegnazione possibile" << endl;
    else
      cout << "Assegnazione impossibile" << endl;
  break;
  }
case 6:
  g.PulisciDischi();
  break;
case 7:
  {
    cout << "Nome disco1: ";
    cin >> nome;
    cout << "Capacita' disco1 : ";
    cin >> dim;
    d = new Disco(dim,nome);
    cout << "Nome disco2: ";
    cin >> nome;
    cout << "Capacita' disco2 : ";
    cin >> dim;
    d2 = new Disco(dim,nome);
    cout << "Dimensione vettore dei file : ";
    cin >> n;
    vector<File*> vf(n);
    for (i = 0; i < n; i++)
      {
        cout << "Nome file : ";
        cin >> nome;
```

```
                cout << "Dimensione : ";
                cin >> dim;
                vf[i] = new File(nome, dim);
              }
            g = *(AssegnaFile(d,d2,vf));
            break;
          }
        }
    }
  while (scelta != 0);
}

Gestore* AssegnaFile(Disco* d1, Disco* d2, vector<File*> v)
{
  unsigned i;
  Gestore* g = new Gestore();

  g->AggiungiDisco(d1);
  g->AggiungiDisco(d2);
  for (i = 0; i < v.size(); i++)
    g->AggiungiFile(v[i]);
  i = 0;
  while(d1->SpazioLibero() >= v[i]->Dimensione())
    {
      g->ScriviFileSuDisco(v[i], d1);
      i++;
    }
  while (i < v.size())
    {
      g->ScriviFileSuDisco(v[i], d2);
      i++;
    }
  return g;
}
```

Soluzione compito del 5 dicembre 2002

Il problema principale di questo compito è scegliere come rappresentare le relazioni tra le tre classi Giocatore, Oggetto e Casella. A questo riguardo si è scelto di rappresentare le relazioni sempre con responsabilità singola.

Per le relazioni "uno-a-molti" si è ovviamente preferito la rappresentazione dal lato dei "molti" in modo da dover memorizzare un singolo puntatore invece di un vettore di puntatori. Quindi, nella classe giocatore, si memorizza la casella in cui si trova e non si memorizza l'insieme dei giocatori presenti in una casella. Analogamente, si memorizza l'oggetto presente su una casella e non l'insieme di caselle che contengono un oggetto.

Riguardo alla relazione "molti-a-molti" tra oggetti e giocatori, si preferisce rappresentarla con responsabilità nella classe Giocatore, in quanto questo è più intuitivo e rende più semplice la gestione della classe Partita.

Per la relazione Adiacenza, l'unica scelta possibile è di realizzarla con un vettore di puntatori alle caselle adiacenti per ogni casella. Si noti che nella classe `Casella` non si è supposto che la relazione sia necessariamente simmetrica, cioè non si impone che se la casella a è nel vettore degli adiacenti della casella b, allora b debba essere nel vettore di a.

Di contro, nella classe `Partita`, quando si inserisce una casella si aggiungono le adiacenze nelle due direzioni (vedi funzione `AggiungiCasella()`).

La classe Partita è la classe principale dell'applicazione, ed ha responsibilità su tutte le sue associazioni, e quindi la classe Partita memorizza dei vettori contenenti i puntatori a tutti gli altri oggetti.

Soluzione esercizio 1

```
// File Gioco.h
#ifndef GIOCO_H
#define GIOCO_H

#include <iostream>
#include <vector>
#include <string>
#include "../Utilities/Random.h"

using namespace std;

class Oggetto
{
public:
  Oggetto(string n, int v) : nome(n), valore(v) {}
  const string nome;
  const int valore;
};

class Casella
{
public:
  Casella(string n, int f) : nome(n) { forza_necessaria = f;
                                       oggetto_presente = NULL; }
  const string nome;
  Oggetto* OggettoPresente() const;
  void MettiOggetto(Oggetto* o) { oggetto_presente = o; }
  Casella* CaselleAdiacenti(unsigned i) const { return adiacenti[i]; }
  bool Raggiungibile(Casella* c) const;
  void RendiRaggiungibile(Casella* c);
  int ForzaNecessaria() const { return forza_necessaria; }
  bool operator==(const Casella& c) const { return nome == c.nome; }
private:
  Oggetto* oggetto_presente;
  vector<Casella*> adiacenti;
  int forza_necessaria;
};

class Giocatore
{
public:
  Giocatore(string n, int f, int m);
  const string nome;
  const int forza;
  const int magia;
  int Energia() const { return energia; }
  void DecrementaEnergia() { energia--; }
  Oggetto* Possiede(int i) const { return oggetti[i]; }
  void PrendiOggetto(Oggetto* o) { oggetti.push_back(o); }
  unsigned NumeroOggetti() const { return oggetti.size(); }
  void PerdiOggetti() { oggetti.clear(); }
  Casella* Posizione() const { return posizione; }
  void PrendiPosizione(Casella* c) { posizione = c; }
  bool operator==(const Giocatore& g) const { return nome == g.nome; }
private:
  int energia;
  vector<Oggetto*> oggetti;
```

```
  Casella* posizione;
};

class Partita
{
  friend ostream& operator<<(ostream& o, const Partita& p);
public:
  Partita(string n) : nome(n) {}
  const string nome;
  void AggiungiGiocatore(Giocatore* g, Casella* c);
  void AggiungiCasella(Casella* c, vector<Casella*> vc);
  void AggiungiOggetto(Oggetto* o, Casella* c);
  unsigned Giocatori() const { return giocatori.size(); }
  unsigned Caselle() const { return caselle.size(); }
  unsigned Oggetti() const { return oggetti.size(); }
  Giocatore* VediGiocatore(int i) const { return giocatori[i]; }
  Casella* VediCasella(int i) const { return caselle[i]; }
  Oggetto* VediOggetto(int i) const { return oggetti[i]; }
  void MuoviGiocatore(Giocatore* g, Casella *c);
  void PrendiOggetto(Giocatore* g);
  void Attacca(Giocatore* g1, Giocatore* g2);
private:
  bool EsisteGiocatore(Giocatore* g) const;
  bool EsisteCasella(Casella* c) const;
  vector<Giocatore*> giocatori;
  vector<Casella*> caselle;
  vector<Oggetto*> oggetti;
};
#endif
```

Soluzione esercizio 2

```
// File Gioco.cpp
#include "Gioco.h"
#include <cassert>

Giocatore::Giocatore(string n, int f, int m)
  : nome(n), forza(f), magia(m)
{
  energia = 10;
  posizione = NULL;
}

Oggetto* Casella::OggettoPresente() const
{
  return oggetto_presente;
}

void Casella::RendiRaggiungibile(Casella* c)
{
  for (unsigned i = 0; i < adiacenti.size(); i++)
    if (*adiacenti[i] == *c)
      return;
  adiacenti.push_back(c);
}

bool Casella::Raggiungibile(Casella* c) const
{
  for (unsigned i = 0; i < adiacenti.size(); i++)
```

```
    if (*adiacenti[i] == *c)
      return true;
  return false;
}

void Partita::AggiungiGiocatore(Giocatore* g, Casella* c)
{
  assert (!EsisteGiocatore(g)&& EsisteCasella(c));
  g->PrendiPosizione(c);
  giocatori.push_back(g);
}

void Partita::AggiungiCasella(Casella* c, vector<Casella*> vc)
{
  assert (!EsisteCasella(c));
  caselle.push_back(c);
  for (unsigned i = 0; i < vc.size(); i++)
    {
      assert(EsisteCasella(vc[i]));
      c->RendiRaggiungibile(vc[i]);
      vc[i]->RendiRaggiungibile(c);
    }
}

void Partita::AggiungiOggetto(Oggetto* o, Casella* c)
{
  assert(EsisteCasella(c));
  c->MettiOggetto(o);
  oggetti.push_back(o);
}

void Partita::MuoviGiocatore(Giocatore* g, Casella *c)
{
  assert(EsisteGiocatore(g) && EsisteCasella(c));
  g->PrendiPosizione(c);
}

void Partita::PrendiOggetto(Giocatore* g)
{
  assert(EsisteGiocatore(g));
  assert(g->Posizione()->OggettoPresente() != NULL);
  assert(g->forza >= g->Posizione()->ForzaNecessaria());
  g->PrendiOggetto(g->Posizione()->OggettoPresente());
}

void Partita::Attacca(Giocatore* g1, Giocatore* g2)
{
  unsigned j;
  assert(EsisteGiocatore(g1) && EsisteGiocatore(g2));
  int ris1 = Random(1,6) + g1->forza;
  int ris2 = Random(1,6) + g2->forza;

  if (ris1 > ris2)
    {
      g1->DecrementaEnergia();
      for (j = 0; j < g1->NumeroOggetti(); j++)
        g2->PrendiOggetto(g1->Possiede(j));
      g1->PerdiOggetti();
    }
```

```
  else
    {
      g2->DecrementaEnergia();
      for (j = 0; j < g2->NumeroOggetti(); j++)
        g1->PrendiOggetto(g2->Possiede(j));
      g2->PerdiOggetti();
    }
}

bool Partita::EsisteGiocatore(Giocatore* g) const
{
  for (unsigned i = 0; i < giocatori.size(); i++)
    if (*giocatori[i] == *g)
      return true;
  return false;
}

bool Partita::EsisteCasella(Casella* c) const
{
  for (unsigned i = 0; i < caselle.size(); i++)
    if (*caselle[i] == *c)
      return true;
  return false;
}

ostream& operator<<(ostream& os, const Partita& p)
{
  unsigned i,j;
  os << "Giocatori: " << endl;
  for (i = 0; i < p.giocatori.size(); i++)
    {
      os << p.giocatori[i]->nome << " (" << p.giocatori[i]->forza
         << ',' << p.giocatori[i]->magia << ',' << p.giocatori[i]->Energia()
         << ") in " << p.giocatori[i]->Posizione()->nome;
      if (p.giocatori[i]->NumeroOggetti() > 0)
        {
          os << " con ";
          for (j = 0; j < p.giocatori[i]->NumeroOggetti(); j++)
            os << p.giocatori[i]->Possiede(j)->nome << ' ';
        }
      os << endl;
    }
  os << endl;
  os << "Caselle: " << endl;
  for (i = 0; i < p.caselle.size(); i++)
    {
      os << p.caselle[i]->nome;
      if (p.caselle[i]->OggettoPresente() != NULL)
        os << " (" << p.caselle[i]->OggettoPresente()->nome << ")";
      os << endl;
    }
  os << endl;
  os << "Adiacenza caselle: " << endl;
  for (i = 0; i < p.caselle.size() - 1; i++)
    for (j = i+1; j < p.caselle.size(); j++)
      if (p.caselle[i]->Raggiungibile(p.caselle[j]))
        os << p.caselle[i]->nome << " <--> " <<  p.caselle[j]->nome << endl;
  os << endl;
  return os;
```

```
}
```

Soluzione esercizio 3 + driver

```
// File main.cpp
#include "Gioco.h"

void MassimaAdiacenza(Partita& p);

int main()
{
  unsigned scelta, i, k, h;
  Giocatore g1("Mago",3,8), g2("Guerriero", 7,2), g3("Elfo",5,5);
  Casella c1("Bosco",3), c2("Prato",3), c3("Castello",5), c4("Prigione",1),
    c5("Foresta",4);
  Oggetto o1("Ascia", 4), o2("Anello", 6), o3("Corda",2);
  Partita p("Sfida dei draghi");

// Crea il tabellone:
//          c2
//        / | \.
//       /  |  \.
//     c1--c5--c4
//      \   |  /
//       \  | /
//         c3

  vector<Casella*> v; // vettore ausiliario per memorizzare di volta in volta
                      // l'insieme delle caselle adiacenti
  p.AggiungiCasella(&c1,v);  // inserisci c1 (senza adiacenze)
  v.push_back(&c1);
  p.AggiungiCasella(&c2,v);  // inserisci c2 adiacente a c1
  p.AggiungiCasella(&c3,v);  // inserisci c3 adiacente a c1
  v[0] = &c2;
  v.push_back(&c3);
  p.AggiungiCasella(&c4,v);  // inserisci c4 adiacente a c2 e c3
  v.push_back(&c1);
  v.push_back(&c4);
  p.AggiungiCasella(&c5,v);  // inserisci c5 adiacente a c1, c2, c3, c4
  // Aggiungi gli oggetti
  p.AggiungiOggetto(&o1,&c4);  // L'ascia e' in prigione
  p.AggiungiOggetto(&o2,&c3);  // L'anello e' nel castello
  p.AggiungiOggetto(&o3,&c5);  // La corda e' nella foresta
  // Aggiungi i giocatori (c1: casella iniziale)
  p.AggiungiGiocatore(&g1, &c1);
  p.AggiungiGiocatore(&g2, &c1);
  p.AggiungiGiocatore(&g3, &c1);

  do
    {
      cout << p;
      cout << "Quale operazione vuoi effettuare?\n"
           << "1: Muovi giocatore\n"
           << "2: Raccogli oggetto\n"
           << "3: Attacca avversario\n"
           << "4: Massima adiacenza\n"
           << "0: Esci\n\n"
           << "Scelta:  ";
      cin >> scelta;
```

```
cout << endl;
switch (scelta)
  {
  case 1:
    {
      cout << "Giocatori: " << endl;
      for (i = 0; i < p.Giocatori(); i++)
        cout << i << " : " << p.VediGiocatore(i)->nome << endl;
      cout << "Quale muovi? : ";
      cin >> k;
      cout << "Caselle: " << endl;
      for (i = 0; i < p.Caselle(); i++)
        {
          if (p.VediGiocatore(k)->Posizione()->Raggiungibile(p.VediCasella(i)))
            cout << i << " : " << p.VediCasella(i)->nome << endl;
        }
      cout << "Casella di arrivo: ";
      cin >> h;
      p.MuoviGiocatore(p.VediGiocatore(k), p.VediCasella(h));
    }
    break;
  case 2:
    {
      cout << "Giocatori: " << endl;
      for (i = 0; i < p.Giocatori(); i++)
        {
          if (p.VediGiocatore(i)->Posizione()->OggettoPresente() != NULL
              && p.VediGiocatore(i)->forza >=  p.VediGiocatore(i)->Posizione()->ForzaNecessari
            cout << i << " : " << p.VediGiocatore(i)->nome << " ("
                 << p.VediGiocatore(i)->Posizione()->OggettoPresente()->nome << ")" << endl;
        }
      cout << "Quale giocatore raccoglie? : ";
      cin >> k;
      p.PrendiOggetto(p.VediGiocatore(k));

    }
    break;
  case 3:
      {
      cout << "Giocatori: " << endl;
      for (i = 0; i < p.Giocatori(); i++)
        cout << i << " : " << p.VediGiocatore(i)->nome << endl;
      cout << "Quale giocatore esegue l'attacco? : ";
      cin >> k;
      cout << "Giocatori nella stessa casella: " << endl;
      for (i = 0; i < p.Giocatori(); i++)
        {
          if (*(p.VediGiocatore(i)->Posizione()) ==
              *(p.VediGiocatore(k)->Posizione()) && i != k)
            cout << i << " : " << p.VediGiocatore(i)->nome << endl;
        }
      cout << "Quale giocatore viene attaccato? : ";
      cin >> h;
      p.Attacca(p.VediGiocatore(k),p.VediGiocatore(h));
    }

    break;
 case 4:
   MassimaAdiacenza(p);
```

```
        break;
       }
    }
  while (scelta != 0);
}


void MassimaAdiacenza(Partita& p)
{
  for (unsigned i = 0; i < p.Caselle() - 1; i++)
    for (unsigned j = i+1; j < p.Caselle(); j++)
      p.VediCasella(i)->RendiRaggiungibile(p.VediCasella(j));
}
```

Soluzione esercizio 4

Il programma stampa l'ultima cifra della matricola aumentata di 1. Infatti, a causa della copia superficiale, gli oggetti `b1` e `b2` condividono lo stesso oggetto di tipo `A`. Quindi quando l'oggetto puntato da `b2` viene modificato, anche quello puntato da `b1` viene modificato.

Una definizione del costruttore di copia che evita la condivisione è la seguente.

```
B::B(const B& b)
{
  p_a = new A(b.S());
}
```

Ovviamente sará necessario ridefinire anche l'operatore di assegnazione. Una versione corretta (non richiesta per il compito) è la seguente.

```
B& operator=(const B& b)
{
  *p_a = *(b.p_a);  // copia il valore puntato, non il puntatore
  return *this;
}
```

Soluzione compito del 18 dicembre 2002

Il questo compito esiste una scelta preferenziale per l'attribuzione della responsabilità delle associazioni. Nella classe `GestioneCinema` si include un vettore di puntatori alla classe `Proiezione`, e nella classe `Proiezione` si includono il film, la sala ed il vettore delle prenotazioni.

Soluzione esercizio 1

```
// File proiezione.h
#ifndef PROIEZIONE_H
#define PROIEZIONE_H

#include <vector>
#include <iostream>
#include "film.h"
#include "sala.h"
#include "orario.h"
#include "prenotazione.h"

using namespace std;
```

```
class Proiezione
{
  friend ostream& operator<<(ostream& os, const Proiezione& p);
public:
  Proiezione(Film* f, Sala* s, Orario inizio, Orario fine);
  Film* FilmProiettato() const { return film; }
  Sala* SalaProiezione() const { return sala; }
  Prenotazione* VediPrenotazione(unsigned i) const { return prenotazioni[i]; }
  unsigned NumeroPrenotazioni() const { return prenotazioni.size(); }
  Orario OraInizio() const { return ora_inizio; }
  Orario OraFine() const { return ora_fine; }
  void AggiungiPrenotazione(Prenotazione* p);
  void CambiaSala(Sala* s);
  void CambiaOrario(Orario i, Orario f);
private:
  Film* film;
  Sala* sala;
  Orario ora_inizio, ora_fine;
  vector<Prenotazione*> prenotazioni;
  unsigned totale_prenotazioni;
};

#endif
```

```
// File cinema.h
#ifndef CINEMA_H
#define CINEMA_H

#include <vector>
#include <iostream>
#include "proiezione.h"

using namespace std;

class GestioneCinema
{
  friend ostream& operator<<(ostream& os, const GestioneCinema& c);
public:
  GestioneCinema(string n);
  void InserisciProiezione(Proiezione* p);
  void CambiaOrario(Proiezione* p, Orario i, Orario o);
  void CambiaSala(Proiezione* p, Sala* s);
  void InserisciPrenotazione(Prenotazione* pn, Proiezione* pi);
  unsigned NumeroProiezioni() const { return proiezioni.size(); }
  Proiezione* VediProiezione(int i) const { return proiezioni[i]; }
private:
  bool VerificaOrario(Orario i, Orario f, Sala* s);
  vector<Proiezione*> proiezioni;
  string nome;
};
#endif
```

Soluzione esercizio 2

Si noti il piccolo artificio utilizzato nella funzione `CambiaOrario()`. Come prima cosa si rende nulla la proiezione corrente mettendo l'inizio e la fine entrambe all'ora 0. Successivamente si verifica che l'orario nuovo sia libero da proiezioni, infine si esegue il cambio. Il primo passo serve per evitare che il cambio di orario sia considerato non fattibile non per una vera sovrapposizione ma per la presenza della proiezione stessa (ad es. se sposto una proiezione di 15m in avanti).

```
// File proiezione.cpp

#include "proiezione.h"
#include <cassert>

Proiezione::Proiezione(Film* f, Sala* s, Orario inizio, Orario fine)
 : ora_inizio(inizio), ora_fine(fine)
{
  film = f;
  sala = s;
  totale_prenotazioni = 0;
}

void Proiezione::AggiungiPrenotazione(Prenotazione* p)
{
  assert(totale_prenotazioni + p->Posti() <= sala->Capienza());
  prenotazioni.push_back(p);
  totale_prenotazioni += p->Posti();
}

void Proiezione::CambiaSala(Sala* s)
{
  assert(totale_prenotazioni <= s->Capienza());
  sala = s;
}

void Proiezione::CambiaOrario(Orario i, Orario f)
{
  ora_inizio = i;
  ora_fine = f;
}

ostream& operator<<(ostream& os, const Proiezione& p)
{
  os << p.film->Titolo() << ", sala " << p.sala->Nome() << ", "
     << p.ora_inizio << "---" << p.ora_fine;
  return os;
}


// File cinema.cpp

#include "cinema.h"
#include <cassert>

GestioneCinema::GestioneCinema(string n)
  : nome(n)
{}

void GestioneCinema::InserisciProiezione(Proiezione* p)
{
  assert(VerificaOrario(p->OraInizio(),p->OraFine(),p->SalaProiezione()));
  proiezioni.push_back(p);
}

void GestioneCinema::CambiaOrario(Proiezione* p, Orario i, Orario o)
{
  p->CambiaOrario(Orario(0,0),Orario(0,0));
  assert(VerificaOrario(i,o,p->SalaProiezione()));
  p->CambiaOrario(i,o);
```

```
}

void GestioneCinema::CambiaSala(Proiezione* p, Sala* s)
{
  assert(VerificaOrario(p->OraInizio(),p->OraFine(),s));
  p->CambiaSala(s);
}

void GestioneCinema::InserisciPrenotazione(Prenotazione* pn, Proiezione* p)
{
  p->AggiungiPrenotazione(pn);
}

bool GestioneCinema::VerificaOrario(Orario inizio, Orario fine, Sala* s)
{
  for (unsigned i = 0; i < proiezioni.size(); i++)
    if (*(proiezioni[i]->SalaProiezione()) == *s)
      if (!(proiezioni[i]->OraFine() <= inizio
            || fine <= proiezioni[i]->OraInizio()))
        return false;
  return true;
}

ostream& operator<<(ostream& os, const GestioneCinema& c)
{
  for (unsigned i = 0; i < c.proiezioni.size(); i++)
    os << *(c.proiezioni[i]) << endl;
  return os;
}
```

Soluzione esercizi 3 e 4 + driver

```
// File main.cpp

#include "cinema.h"

bool EsisteFilmTroppoLungo(const GestioneCinema& c);

int ContaClientiFedeli(const GestioneCinema& c, vector<string> v, int k);

int main()
{  // driver incompleto, e non dotato di menu
  Film f1("Zelig",123), f2("La rosa purpurea del cairo",90),
    f3("Crimini e misfatti", 200), f4("Anna e le sue sorelle", 93);
  Sala s1("A",200), s2("B",150);
  Proiezione p1(&f1,&s1,Orario(17,00),Orario(19,20));
  Proiezione p2(&f1,&s2,Orario(17,00),Orario(19,20));
  Proiezione p3(&f2,&s1,Orario(15,00),Orario(17,00));
  Proiezione p4(&f2,&s2,Orario(19,40),Orario(21,20));
  Proiezione p5(&f3,&s1,Orario(19,30),Orario(21,30));
  Proiezione p6(&f4,&s2,Orario(22,00),Orario(23,50));
  GestioneCinema c("Multisala verdi");

  c.InserisciProiezione(&p1);
  c.InserisciProiezione(&p2);
  c.InserisciProiezione(&p3);
  c.InserisciProiezione(&p4);
  c.InserisciProiezione(&p5);
  c.InserisciProiezione(&p6);
```

```
  cout << c << endl;
  if (EsisteFilmTroppoLungo(c))
    cout << "Esiste un film troppo lungo" << endl;
  else
    cout << "Tutti i film sono a posto con la lunghezza" << endl;
}

bool EsisteFilmTroppoLungo(const GestioneCinema& c)
{
  for (unsigned i = 0; i < c.NumeroProiezioni(); i++)
    {
      Proiezione* p = c.VediProiezione(i);
      if (p->FilmProiettato()->Durata() > p->OraFine() - p->OraInizio())
        return true;
    }
  return false;
}

int ContaClientiFedeli(const GestioneCinema& c, vector<string> v, int k)
{
  int conta_prenotazioni, conta_persone = 0;
  for (unsigned i = 0; i < v.size(); i++)
    {
      conta_prenotazioni = 0;
      for (unsigned j = 0; j < c.NumeroProiezioni(); j++)
        {
          Proiezione* p = c.VediProiezione(i);
          for (unsigned h = 0; h < p->NumeroPrenotazioni(); h++)
            if (p->VediPrenotazione(h)->NomePrenotante() == v[i])
              conta_prenotazioni += p->VediPrenotazione(h)->Posti();
        }
      if (conta_prenotazioni > k)
        conta_persone++;
    }
  return conta_persone;
}
```

Classi e funzioni predefinite

```
// File film.h
#ifndef FILM_H
#define FILM_H

#include <string>

using namespace std;

class Film
{
  friend bool operator==(const Film& f1, const Film& f2);
public:
  Film(string t, int d);
  int Durata() const { return durata; }
  string Titolo() const { return titolo; }
private:
  string titolo;
  int durata; // in minuti
};
```

```
#endif

// File film.cpp

#include "film.h"

Film::Film(string t, int d)
  : titolo(t)
{
  durata = d;
}

bool operator==(const Film& f1, const Film& f2)
{
  return f1.titolo == f2.titolo;
}


// File orario.h
#ifndef ORARIO_H
#define ORARIO_H

#include <iostream>

using namespace std;

class Orario
{
  friend bool operator<(const Orario& o1, const Orario& o2);
  friend bool operator<=(const Orario& o1, const Orario& o2);
  friend bool operator==(const Orario& o1, const Orario& o2);
  friend int operator-(const Orario& o1, const Orario& o2);  // differenza in minuti
  friend ostream& operator<<(ostream& os, const Orario& o);
public:
  Orario(int o, int m);
  int Ore() const { return ore; }
  int Minuti() const { return minuti; }
private:
  int ore, minuti;
};

#endif

// File orario.cpp

#include "orario.h"

bool operator<(const Orario& o1, const Orario& o2)
{
  return o1.ore < o2.ore || (o1.ore == o2.ore && o1.minuti < o2.minuti);
}

bool operator<=(const Orario& o1, const Orario& o2)
{
  return o1.ore < o2.ore || (o1.ore == o2.ore && o1.minuti <= o2.minuti);
}

bool operator==(const Orario& o1, const Orario& o2)
{
  return o1.ore == o2.ore && o1.minuti == o2.minuti;
```

```
}

int operator-(const Orario& o1, const Orario& o2)
{
  return (o1.ore - o2.ore) * 60 + o1.minuti - o2.minuti;
}

ostream& operator<<(ostream& os, const Orario& o)
{
  os << o.ore << ':';
  if (o.minuti < 9) os << '0'; // per una stampa piu' gradevole
  os << o.minuti;
  return os;
}

Orario::Orario(int o, int m)
{
  ore = o;
  minuti = m;
}


// File sala.h
#ifndef SALA_H
#define SALA_H

#include <string>

using namespace std;

class Sala
{
  friend bool operator==(const Sala& s1, const Sala& s2);
public:
  Sala(string n, int c);
  int Capienza() const { return capienza; }
  string Nome() const { return nome; }
private:
  string nome;
  int capienza;
};

#endif

// File sala.cpp

#include "sala.h"

Sala::Sala(string n, int c)
  : nome(n)
{
  capienza = c;
}

bool operator==(const Sala& s1, const Sala& s2)
{
  return s1.nome == s2.nome;
}

// File prenotazione.h
```

```
#ifndef PRENOTAZIONE_H
#define PRENOTAZIONE_H

#include <string>

using namespace std;

class Prenotazione
{
public:
  Prenotazione(string n, unsigned posti);
  string NomePrenotante() const { return nome_prenotante; }
  unsigned Posti() const { return posti; }
private:
  string nome_prenotante;
  unsigned posti;
};

#endif
```

```
// File prenotazione.cpp

#include "prenotazione.h"

Prenotazione::Prenotazione(string n, unsigned p)
  : nome_prenotante(n)
{
  posti = p;
}
```

Soluzione compito del 7 luglio 2003

La definizione fornita per le classi `Medico` e `Assistito` ci impone di dare la responsabilità dell'associazione tra Visita e queste due classi alla classe `Visita`. Tale classe sarà dotata quindi di due puntatori al medico e all'assistito, rispettivamente.

La classe principale `Asl` sarà invece dotata di tre vettore di puntatori, per memorizzare i riferimenti a tutti gli oggetti coinvolti nell'applicazione.

Soluzione esercizio 1

```
// File asl.h
#ifndef ASL_H
#define ASL_H

#include <vector>
#include <string>
#include <iostream>
#include "Data.h"

using namespace std;
const int ASSENTE = -1;

class Medico
{
  friend ostream& operator<<(ostream& os, const Medico& m);
public:
  Medico(string n, string s, int a);
```

```
  string Nome() const {return nome; }
  string Specializzazione() const {return spec; }
  void CambiaSpecializzazione(string s) { spec = s; }
  int Anzianita() const { return anzianita; }
private:
  string nome, spec;
  int anzianita;
};

class Assistito
{
  friend ostream& operator<<(ostream& os, const Assistito& a);
public:
  Assistito(string n, int nt);
  string Nome() const {return nome; }
  int NumTessera() const { return num_tessera; }
private:
  string nome;
  int num_tessera;
};

class Visita
{
  friend ostream& operator<<(ostream& os, const Visita& v);
  friend bool operator==(const Visita& v1, const Visita& v2);
public:
  Visita(string t, string f, int du, Data da, Medico *m, Assistito *a);
  Visita(string f, Data d, Medico *m, Assistito *a);
  string Tipo() const { return tipo; }
  void CambiaTipo(string t) { tipo = t; }
  string Farmaco() const { return farmaco; }
  int Durata() const { return durata; }
  Data DataVisita() const { return data; }
  Medico* Visitante() const { return p_med; }
  Assistito* Visitato() const { return p_ass; }
private:
  string tipo, farmaco;
  int durata;
  Data data;
  Medico* p_med;
  Assistito* p_ass;
};

class Asl
{
  friend ostream& operator<<(ostream& os, const Asl& v);
public:
  Asl(string n, float b);
  void InserisciVisita(Visita* v);
  void CreaVisita(Medico* m, Assistito* a, Data d, string f);
  void InserisciAssistito(Assistito* a);
  void InserisciMedico(Medico* m);
  void CorreggiSpecializzazione(Medico* m, string s);
  string Nome() const { return nome; }
  float Budget() const { return budget; }
  unsigned NumMedici() const { return medici.size(); }
  unsigned NumAssistiti() const { return assistiti.size(); }
  unsigned NumVisite() const { return visite.size(); }
  Medico* VediMedico(unsigned i) const { return medici[i]; }
```

```
  Assistito* VediAssistito(unsigned i) const { return assistiti[i]; }
  Visita* VediVisita(unsigned i) const { return visite[i]; }
private:
  int CercaVisita(Visita* v);
  int CercaMedico(Medico* m);
  int CercaAssistito(Assistito* a);
  string nome;
  float budget;
  vector<Medico*> medici;
  vector<Assistito*> assistiti;
  vector<Visita*> visite;
};
#endif
```

Soluzione esercizi 2 e 4

```
// File asl.cpp
#include "asl.h"
#include <cassert>

ostream& operator<<(ostream& os, const Medico& m)
{
  os << m.nome << " - " << m.spec << " (" << m.anzianita << ") ";
  return os;
}

Medico::Medico(string n, string s, int a)
  : nome(n), spec(s), anzianita(a)
{}

ostream& operator<<(ostream& os, const Assistito& a)
{
  os << a.nome << " - " << a.num_tessera;
  return os;
}

Assistito::Assistito(string n, int nt)
  : nome(n), num_tessera(nt)
{}

ostream& operator<<(ostream& os, const Visita& v)
{
  os << v.p_med->Nome() << " visita " << v.p_ass->Nome() << " " << v.durata
     << " " << v.data << " " << v.tipo << " (" << v.farmaco << ")";
  return os;
}

Visita::Visita(string t, string f, int du, Data da, Medico *m, Assistito *a)
  : tipo(t), farmaco(f), durata(du), data(da), p_med(m), p_ass(a)
{}

Visita::Visita(string f, Data d, Medico *m, Assistito *a)
  : tipo(m->Specializzazione()), farmaco(f),
    durata(30), data(d), p_med(m), p_ass(a)
{}

Asl::Asl(string n, float b)
:  nome(n), budget(b)
{}
```

```
int Asl::CercaVisita(Visita* v)
{
  for (unsigned i = 0; i < visite.size(); i++)
    if (*v == *visite[i])
      return i;
  return ASSENTE;
}

int Asl::CercaAssistito(Assistito* a)
{
  for (unsigned i = 0; i < assistiti.size(); i++)
    if (a->NumTessera() == assistiti[i]->NumTessera())
      return i;
  return ASSENTE;
}

int Asl::CercaMedico(Medico* m)
{
  for (unsigned i = 0; i < medici.size(); i++)
    if (m->Nome() == medici[i]->Nome())
      return i;
  return ASSENTE;
}

bool operator==(const Visita& v1, const Visita& v2)
{
  return v1.Visitante()->Nome() == v2.Visitante()->Nome()
    && v1.Visitato()->NumTessera() == v2.Visitato()->NumTessera()
    && v1.Tipo() == v2.Tipo()
    && v1.DataVisita() == v2.DataVisita();
}

void Asl::InserisciVisita(Visita* v)
{
  assert(CercaMedico(v->Visitante()) != ASSENTE);
  assert(CercaAssistito(v->Visitato()) != ASSENTE);
  assert(CercaVisita(v) == ASSENTE);
  visite.push_back(v);
}

void Asl::CreaVisita(Medico* m, Assistito* a, Data d, string f)
{
  Visita* v = new Visita(f,d,m,a);
  InserisciVisita(v);
}

void Asl::InserisciAssistito(Assistito* a)
{
  assert(CercaAssistito(a) == ASSENTE);
  assistiti.push_back(a);
}

void Asl::InserisciMedico(Medico* m)
{
  if (CercaMedico(m) == ASSENTE)
    medici.push_back(m);
}
```

```
void Asl::CorreggiSpecializzazione(Medico* m, string s)
{
  int i;
  string s1;

  i = CercaMedico(m);
  assert (i != ASSENTE);
  s1 = medici[i]->Specializzazione();
  assert (s1 != s);
  medici[i]->CambiaSpecializzazione(s);
  for (unsigned j = 0; j < visite.size(); j++)
    {
      if (visite[j]->Visitante()->Nome() == m->Nome()
          && visite[j]->Tipo() == s1)
        visite[j]->CambiaTipo(s);
    }
}

ostream& operator<<(ostream& os, const Asl& a)
{
  unsigned i;
  os << "ASL " << a.nome << " (budget " << a.budget << ")" << endl;
  os << "Medici: " << endl;
  for (i = 0; i < a.medici.size(); i++)
    os << *(a.medici[i]) << endl;
  os << endl;
  os << "Assistiti: " << endl;
  for (i = 0; i < a.assistiti.size(); i++)
    os << *(a.assistiti[i]) << endl;
  os << endl;
  os << "Visite: " << endl;
  for (i = 0; i < a.visite.size(); i++)
    os << *(a.visite[i]) << endl;
  os << endl;
  return os;
}
```

Soluzione esercizi 3 + driver

```
#include "asl.h"

float Rapporto(const Asl& asl, Medico* m);
string MedicoFedele(const Asl& asl);

int main()
{
  Data oggi(28,3,2003), ieri(27,3,2003);
  Medico m1("Mario Rossi","Cardiologia",10);
  Medico m2("Marta Verdi","Pediatria",20);
  Assistito a1("Andrea Bianchi",1);
  Assistito a2("Franco Neri",2);
  Assistito a3("Irene Gialli",3);
  Visita v1("Otorino", "Anauran", 20, oggi, &m1, &a1);
  Visita v2("Prozac", oggi, &m1, &a1);
  Visita v3("Niente", ieri, &m1, &a3);
  Asl u1("Udine 1", 1000);

  u1.InserisciAssistito(&a1);
  u1.InserisciAssistito(&a2);
```

```
  u1.InserisciAssistito(&a3);
  u1.InserisciMedico(&m1);
  u1.InserisciMedico(&m2);
  u1.InserisciVisita(&v1);
  u1.InserisciVisita(&v2);
  u1.InserisciVisita(&v3);
  u1.CreaVisita(&m1,&a1,ieri,"Aspirina");
  cerr << u1 << endl;
  u1.CorreggiSpecializzazione(&m1,"Medicina Generale");
  cerr << u1 << endl;
}

float Rapporto(const Asl& asl, Medico* m)
{ // funzione ausiliaria di MedicoFedele
  int visite_totali = 0, visite_specialistiche = 0;
  for (unsigned i = 0; i < asl.NumVisite(); i++)
    if (asl.VediVisita(i)->Visitante()->Nome() == m->Nome())
      {
        visite_totali++;
        if (asl.VediVisita(i)->Tipo() == m->Specializzazione())
          visite_specialistiche++;
      }
  return float(visite_specialistiche)/visite_totali;
}

string MedicoFedele(const Asl& asl)
{
  string nome_max;
  float rapporto_max = 0, rapporto;
  for (unsigned i = 0; i < asl.NumMedici(); i++)
    {
      rapporto = Rapporto(asl,asl.VediMedico(i));
      if (rapporto > rapporto_max)
        nome_max = asl.VediMedico(i)->Nome();
    }
  return nome_max;
}
```

Soluzione compito del 2 settembre 2003

Soluzione esercizio 1

Per la codifica degli stati nel programma si è scelto di usare una variabile intera, come spiegato nei commenti del file `Candidato.h`. Per ogni candidato, viene memorizzato oltre al suo stato, anche il vettore delle prove che ha eseguito. In questo vettore vengono mantenute solo le prove *valide*, cioè quelle significative per l'ammissione; conseguentemente, quando un candidato torna indietro nel percorso le prove non più valide vengono eliminate (ad es. si veda la funzione `DecisioneFinale()`).

Si noti inoltre che per rappresentare l'associazione tra Candidato e Prova si è scelto di utilizzare un vettore di oggetti. Questa scelta è possibile in quando gli oggetti della classe Prova non hanno altri legami (e quindi possono essere "interni" all'oggetto di tipo Candidato)

File `Candidato.h`

```
// File Candidato.h
#ifndef CANDIDATO_H
#define CANDIDATO_H
```

```
#include "../2003-06-23/Data.h"
#include <vector>

using namespace std;
const unsigned SOGLIA_MAESTRO = 25;
// Codifica dello stato:
// 0: Matricola
// 1: Iniziato
// 2: Allievo
// 3: Futuro Esperto
// 4: Futuro Maestro
// 5: Non ammesso
// 6: Ammesso esperto
// 7: Ammesso maestro

class Prova
{
public:
  Prova(unsigned v, Data d) : voto(v), data(d) {}
  unsigned voto;
  Data data;
};

class Candidato
{
  friend ostream& operator<<(ostream& os, const Candidato& c);
public:
  Candidato(string n);
  void EseguiProva(int voto, Data data);
  void DecisioneFinale(string decisione);
  int Stato() const { return stato; }
  string Nome() const { return nome; }
  int VotoProva(unsigned i) const { return prove[i].voto; }
  Data DataProva(unsigned i) const { return prove[i].data; }
private:
  string nome;
  unsigned stato;
  vector<Prova> prove;
};

#endif
```

File `Associazione.h`

```
// File Associazione.h
#ifndef ASSOCIAZIONE_H
#define ASSOCIAZIONE_H
#include "Candidato.h"

class Membro
{
public:
  Membro(string nome, string qualifica);
  string Nome() const { return nome; }
  string Qualifica() const { return qualifica; }
private:
  string nome;
  string qualifica;
};
```

```
class Associazione
{
  friend ostream& operator<<(ostream& os, const Associazione& a);
public:
  Associazione(string nome);
  ~Associazione();
  void InserisciCandidato(string nome);
  void EseguiProve(Data data, vector<unsigned> voti);
  void RegistraAccettazioni(vector<bool> accettazioni);
  void NominaNuoviMembri();
  unsigned Candidati() const { return candidati.size(); }
  unsigned CandidatiInProva() const;
  unsigned Membri() const { return membri.size(); }
  Membro* VediMembro(unsigned i) const { return membri[i]; }
  Candidato* VediCandidato(unsigned i) const { return candidati[i]; }
private:
  vector<Membro*> membri;
  vector<Candidato*> candidati;
  string nome;
};
#endif
```

Soluzione esercizio 2

File Candidato.cpp

```
// File Candidato.cpp

#include "Candidato.h"
#include <cassert>

Candidato::Candidato(string n)
  : nome(n), stato(0)
{}

void Candidato::EseguiProva(int voto, Data data)
{
  assert(stato <= 2);
  if (stato == 0)
    if (voto < 6)
      stato = 5;
    else
      {
        stato = 1;
        prove.push_back(Prova(voto, data));
      }
  else if ((stato == 1 && voto < 7) || (stato == 2 && voto < 8))
    {
      stato = 0;
      prove.clear();
    }
  else if (stato == 1 && voto >= 7)
    {
      stato = 2;
      prove.push_back(Prova(voto, data));
    }
  else if (stato == 2 && voto >= 8)
    {
      unsigned totale;
```

```
      prove.push_back(Prova(voto, data));
      totale = prove[0].voto + prove[1].voto + prove[2].voto;
      if (totale >= SOGLIA_MAESTRO)
        stato = 4;
      else
        stato = 3;
    }
}

void Candidato::DecisioneFinale(string decisione)
{
  assert(stato == 3 || (stato == 4 && decisione == "Accetta"));
  if (stato == 3)
    if (decisione == "Accetta")
      stato = 6;
    else // decisione == "Rifiuta"
      {
        stato = 1;
        prove.pop_back(); // elimina la terza prova
        prove.pop_back(); // elimina la seconda prova

      }
  else // stato == 4
    stato = 7;
}

ostream& operator<<(ostream& os, const Candidato& c)
{
  os << c.nome << "  (";
  if (c.stato == 0)
    os << "Matricola";
  else if (c.stato == 1)
    os << "Iniziato";
  else if (c.stato == 2)
    os << "Allievo";
  else if (c.stato == 3)
    os << "Candidato Esperto";
  else if (c.stato == 4)
    os << "Candidato Maestro";
  else if (c.stato == 5)
    os << "Non ammesso";
  else if (c.stato == 6)
    os << "Ammesso Esperto";
  else // c.stato == 7
    os << "Ammesso Maestro";
  os << ")";
  if (c.stato != 0 && c.stato != 5)
    {
      os  << endl << "Prove :" << endl;
      for (unsigned i = 0; i < c.stato && i < 3; i++)
        os << i+1 << ". Voto : " << c.prove[i].voto
           << " (fatta il " << c.prove[i].data << ")" << endl;
    }
  return os;
}
```

File `Associazione.cpp`

```
// File Associazione.cpp
```

```
#include "Associazione.h"

Membro::Membro(string n, string q)
  : nome(n), qualifica(q) {}

Associazione::Associazione(string n)
  : nome(n) {}

Associazione::~Associazione()
{
  unsigned i;
  for (i = 0; i < membri.size(); i++)
    delete membri[i];
  for (i = 0; i < candidati.size(); i++)
    delete candidati[i];
}

void Associazione::InserisciCandidato(string n)
{
  Candidato* c = new Candidato(n);
  candidati.push_back(c);
}

unsigned Associazione::CandidatiInProva() const
{
  unsigned conta = 0;
  for (unsigned i = 0; i < candidati.size(); i++)
    if (candidati[i]->Stato() <= 2)
      conta++;
  return conta;
}

void Associazione::EseguiProve(Data data, vector<unsigned> voti)
{
  unsigned i = 0, j = 0;
  for (i = 0; i < candidati.size(); i++)
    {
      if (candidati[i]->Stato() <= 2)
        {
          candidati[i]->EseguiProva(voti[j], data);
          j++;
        }
    }
}

void Associazione::RegistraAccettazioni(vector<bool> accettazioni)
{
  unsigned i, j = 0;
  for (i = 0; i < candidati.size(); i++)
    if (candidati[i]->Stato() == 4)
      {
        candidati[i]->DecisioneFinale("Accetta");
      }
    else if (candidati[i]->Stato() == 3)
      {
        if (accettazioni[j])
          candidati[i]->DecisioneFinale("Accetta");
        else
```

```
          candidati[i]->DecisioneFinale("Rifiuta");
        j++;
      }
}

void Associazione::NominaNuoviMembri()
{
  unsigned i = 0;
  string qualifica;
  while (i < candidati.size())
    {
      if (candidati[i]->Stato() == 6 || candidati[i]->Stato() == 7)
        {
          if (candidati[i]->Stato() == 6)
            qualifica = "Esperto";
          else
            qualifica = "Maestro";
          Membro* m = new Membro(candidati[i]->Nome(), qualifica);
          membri.push_back(m);
          delete candidati[i];
          candidati.erase(candidati.begin() + i);
        }
      else if (candidati[i]->Stato() == 5)
        {
          delete candidati[i];
          candidati.erase(candidati.begin() + i);
        }
      else
        i++;
    }
}

ostream& operator<<(ostream& os, const Associazione& a)
{
  unsigned i;
  os << "Membri : " << a.Membri() << endl;
  for (i = 0; i < a.Membri(); i++)
    os << a.membri[i]->Nome() << "   " << a.membri[i]->Qualifica() << endl;

  os << "Candidati : " << a.Candidati() << endl;
  for (i = 0; i < a.Candidati(); i++)
    os << *(a.candidati[i]) << endl;
  return os;
}
```

Soluzione esercizi 3 (driver)

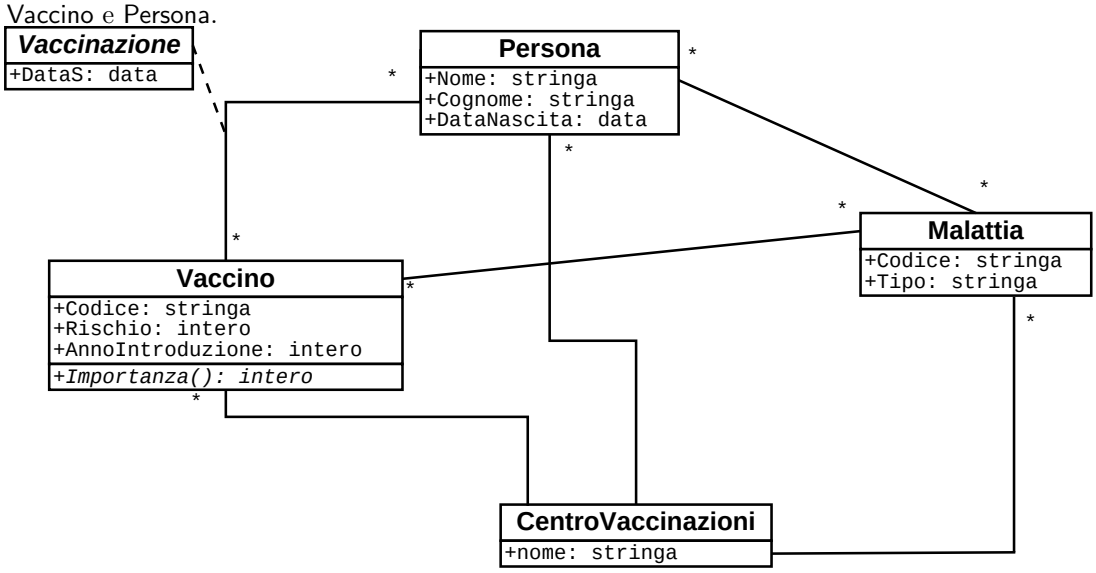
```
// File main.cpp

#include "Associazione.h"

int main()
{
  Associazione a("Amici della pelota");
  int scelta;
  do
    {
      cout << a;
      cout << "Quale operazione vuoi effettuare?\n"
```

```
           << "1: Aggiungi candidato\n"
           << "2: Esegui prove\n"
           << "3: Registra Accettazioni\n"
           << "4: Nomina nuovi membri\n"
           << "0: Esci\n\n"
           << "Scelta:  ";
      cin >> scelta;
      cout << endl;
      switch (scelta)
        {
        case 1:
          {
            string nome;
            cout << "Nome candidato : ";
            cin >> nome;
            a.InserisciCandidato(nome);
          }
          break;
        case 2:
          {
            Data d;
            unsigned dim = a.CandidatiInProva();
            vector<unsigned> v(dim);
            cout << "Data : ";
            cin >> d;
            cout << "Voti (" << dim << " candidati in prova) : ";
            for (unsigned i = 0; i < dim; i++)
              cin >> v[i];
            a.EseguiProve(d,v);
          }
          break;
        case 3:
            {
            vector<bool> v;
            string s;
            cout << "Vettore accettazioni: ";
            for (unsigned i = 0; i < a.Candidati(); i++)
              if (a.VediCandidato(i)->Stato() == 3)
                {
                  cout << a.VediCandidato(i)->Nome() << " ? (si/no) : ";
                  cin >> s;
                  if (s == "si")
                    v.push_back(true);
                  else
                    v.push_back(false);
                }
            a.RegistraAccettazioni(v);
          }
          break;
        case 4:
          a.NominaNuoviMembri();
        }
    }
  while (scelta != 0);
}
```

Soluzione compito del 16 settembre 2003

Soluzione esercizio 1

Si noti che Vaccinazione non è una classe (come molti studenti hanno scritto all'esame), ma una relazione tra Vaccino e Persona.



Soluzione esercizio 2

Per risolvere questo esercizio si pone il problema di a chi attribuire la responsabilitelle relazioni tra la classi Persona, Vaccino e Malattia.

La soluzione ci viene suggerita dal testo, in cui si legge che:

- dal vaccino vogliamo risalire alle persone a cui è stato somministrato, e non viceversa;
- dalla malattia vogliamo risalire alle persone che l'hanno contratta, e non viceversa;
- dal vaccino voglia risalire alle malattie che previene, e non viceversa.

Il base alle precedenti osservazioni, si sono decise le responsabilità che si evincono dalle seguenti definizioni di classe.

```
// File vaccinazioni.h
#ifndef VACCINAZIONI_H
#define VACCINAZIONI_H

#include <iostream>
#include <string>
#include <vector>
#include "../2003-07-07/Data.h"

using namespace std;

class Persona
{
  friend ostream& operator<<(ostream& os, const Persona& p);
public:
  Persona(string n, string c, Data d);
  string Nome() const { return nome; }
  string Cognome() const { return cognome; }
  Data DataNascita() const { return data_nascita; }
  bool operator==(const Persona& p);
private:
  string nome;
```

```
  string cognome;
  Data data_nascita;
};

class Malattia
{
  friend ostream& operator<<(ostream& os, const Malattia& m);
public:
  Malattia(string c, string t);
  string Codice() const { return codice; }
  string Tipo() const { return tipo; }
  Persona* VediMalato(int i) const { return malati[i]; }
  void InserisciMalato(Persona* p);
  int NumMalati() const { return malati.size(); }
private:
  int CercaMalato(Persona *p) const;
  vector <Persona*> malati;
  string codice;
  string tipo;
};

class Vaccinazione
{
public:
  Vaccinazione(Persona* p, Data d);
  Persona* Vaccinato() const { return vaccinato; }
  Data DataVaccinazione() const { return data_somministrazione; }
private:
  Persona* vaccinato;
  Data data_somministrazione;
};

class Vaccino
{
  friend ostream& operator<<(ostream& os, const Vaccino& v);
public:
  Vaccino(string n);
  string Nome() const { return nome; }
  Malattia* VediMalattia(int i) const { return malattie[i]; }
  Vaccinazione VediVaccinazione(int i) const { return vaccinazioni[i]; }
  void InserisciMalattia(Malattia *m);
  void InserisciVaccinazione(Persona* p, Data d);
  int Vaccinazioni() const { return vaccinazioni.size(); }
  int Malattie() const { return malattie.size(); }
  float Importanza() const;
private:
  int CercaMalattia(Malattia *m) const;
  string nome;
  vector<Malattia*> malattie;
  vector<Vaccinazione> vaccinazioni;
};

class CentroVaccinazioni
{
  friend ostream& operator<<(ostream& os, const CentroVaccinazioni& cv);
public:
  CentroVaccinazioni(string n);
  int Persone() const { return pazienti.size(); }
  int Malattie() const { return malattie.size(); }
```

```
  int Vaccini() const { return vaccini.size(); }
  Persona* VediPaziente(int i) const { return pazienti[i]; }
  Malattia* VediMalattia(int i) const { return malattie[i]; }
  Vaccino* VediVaccino(int i) const { return vaccini[i]; }
  void InserisciPaziente(Persona *);
  void InserisciMalattia(Malattia *);
  void InserisciVaccino(Vaccino *);
  void InserisciVaccinazione(Vaccino*, Persona*, Data d);
private:
  int CercaPaziente(Persona *p) const;
  int CercaMalattia(Malattia *m) const;
  int CercaVaccino(Vaccino *v) const;

  string nome;
  vector<Persona*> pazienti;
  vector<Malattia*> malattie;
  vector<Vaccino*> vaccini;
};
#endif
```

Soluzione esercizio 3

```
// File vaccinazioni.cpp
#include "vaccinazioni.h"
#include <cassert>

Persona::Persona(string n, string c, Data d)
{
  nome = n;
  cognome = c;
  data_nascita = d;
}

bool Persona::operator==(const Persona& p)
{
  return nome == p.nome && cognome == p.cognome;
}

ostream& operator<<(ostream& os, const Persona& p)
{
  os << p.nome << " " << p.cognome << " " << p.data_nascita;
  return os;
}

Malattia::Malattia(string c, string t)
{
  codice = c;
  tipo = t;
}

void Malattia::InserisciMalato(Persona* p)
{
  assert (CercaMalato(p) == -1);
  malati.push_back(p);
}

int Malattia::CercaMalato(Persona *p) const
{
  for (unsigned i = 0; i < malati.size(); i++)
```

```
    if (*(malati[i]) == *p)
      return i;
  return -1;
}

ostream& operator<<(ostream& os, const Malattia& m)
{
  unsigned i;
  os << m.codice << " " << m.tipo <<endl;
  os << "Elenco malati:" << endl;
  for (i = 0; i < m.malati.size(); i++)
    os << "   " << *(m.malati[i]) << endl;
  return os;
}

Vaccinazione::Vaccinazione(Persona* p, Data d)
{
  vaccinato = p;
  data_somministrazione = d;
}

Vaccino::Vaccino(string n)
{
  nome = n;
}

void Vaccino::InserisciMalattia(Malattia *m)
{
  assert (CercaMalattia(m) == -1);
  malattie.push_back(m);
}

void Vaccino::InserisciVaccinazione(Persona *p, Data d)
{
  Vaccinazione vac(p,d);
  vaccinazioni.push_back(vac);
}

int Vaccino::CercaMalattia(Malattia *m) const
{
  for (unsigned i = 0; i < malattie.size(); i++)
    if (malattie[i]->Codice() == m->Codice())
      return i;
  return -1;
}

float Vaccino::Importanza() const
{
  int tot_malati = 0;
  if (malattie.size() == 0)
    return 0;
  else
    {
      for (unsigned i = 0; i < malattie.size(); i++)
        tot_malati += malattie[i]->NumMalati();
      return float(tot_malati)/malattie.size();
    }
}
```

```
ostream& operator<<(ostream& os, const Vaccino& v)
{
  unsigned i;
  os << v.nome << " (" << v.Importanza() << ")" << endl;
  os << "Elenco malattie:" << endl;
  for (i = 0; i < v.malattie.size(); i++)
    os << "   " << *(v.malattie[i]) << endl;
  os << "Elenco vaccinazioni:" << endl;
  for (i = 0; i < v.vaccinazioni.size(); i++)
    os << "   " << *(v.vaccinazioni[i].Vaccinato()) << " (il " << v.vaccinazioni[i].DataVaccinazione
  return os;
}

CentroVaccinazioni::CentroVaccinazioni(string n)
{
  nome = n;
}

void CentroVaccinazioni::InserisciPaziente(Persona* p)
{
  assert(CercaPaziente(p) == -1);
  pazienti.push_back(p);
}

void CentroVaccinazioni::InserisciMalattia(Malattia* m)
{
  assert(CercaMalattia(m) == -1);
  malattie.push_back(m);
}

void CentroVaccinazioni::InserisciVaccino(Vaccino* v)
{
  assert(CercaVaccino(v) == -1);
  vaccini.push_back(v);
}

void CentroVaccinazioni::InserisciVaccinazione(Vaccino* v, Persona *p, Data d)
{
  int i;
  i = CercaVaccino(v);
  assert(i != -1);
  vaccini[i]->InserisciVaccinazione(p,d);
}

int CentroVaccinazioni::CercaPaziente(Persona *p) const
{
  for (unsigned i = 0; i < pazienti.size(); i++)
    if (*(pazienti[i]) == *p)
      return i;
  return -1;
}

int CentroVaccinazioni::CercaMalattia(Malattia *m) const
{
  for (unsigned i = 0; i < malattie.size(); i++)
    if (malattie[i]->Codice() == m->Codice())
      return i;
  return -1;
}
```

```
int CentroVaccinazioni::CercaVaccino(Vaccino *v) const
{
  for (unsigned i = 0; i < vaccini.size(); i++)
    if (vaccini[i]->Nome() == v->Nome())
      return i;
  return -1;
}

ostream& operator<<(ostream& os, const CentroVaccinazioni& cv)
{
  unsigned i;
  os << "Centro vaccinazioni: " << cv.nome << endl;
  os << "---Pazienti: " << endl;
  for (i = 0; i < cv.pazienti.size(); i++)
    os << *(cv.pazienti[i]) << endl;
  os << "---Malattie: " << endl;
  for (i = 0; i < cv.malattie.size(); i++)
    os << *(cv.malattie[i]) << endl;
  os << "---Vaccini: " << endl;
  for (i = 0; i < cv.vaccini.size(); i++)
    os << *(cv.vaccini[i]) << endl;
  return os;
}
```

Soluzione esercizi 4 + driver

```
// File main.cpp
#include "vaccinazioni.h"

vector<Persona*> PersoneVaccinate(Data d, unsigned n, const vector<Vaccino*>& v);

int main()
{
  CentroVaccinazioni cv("asl1");
  Persona* p;
  Malattia* m;
  Vaccino* v;
  Data data;
  string nome, cognome, tipo;
  int scelta, i, num_persona, num_vaccino, num_malattia;

  do
    {
      cout << cv;
      cout << "Quale operazione vuoi effettuare?\n"
           << "1: Inserimento persona\n"
           << "2: Inserimento malattia\n"
           << "3: Inserimento vaccino\n"
           << "4: Inserimento vaccinazione (persona, vaccino)\n"
           << "5: Inserimento contagio (persona, malattia)\n"
           << "6: Inserimento cura (vaccino, malattia)\n"
           << "7: Persone vaccinate\n"
           << "0: Esci\n\n"
           << "Scelta:  ";
      cin >> scelta;
      cout << endl;
      switch (scelta)
        {
```

```
 case 1:
   cout << "Nome e cognome: ";
   cin >> nome >> cognome;
   cout << "Data di nascita: ";
   cin >> data;
   p = new Persona(nome,cognome,data);
   cv.InserisciPaziente(p);
   break;
 case 2:
   cout << "Nome : ";
   cin >> nome;
   cout << "Tipo : ";
   cin >> tipo;
   m = new Malattia(nome,tipo);
   cv.InserisciMalattia(m);
   break;
case 3:
   cout << "Nome : ";
   cin >> nome;
   v = new Vaccino(nome);
   cv.InserisciVaccino(v);
   break;
 case 4:
   cout << "Persone presenti : " << endl;
   for (i = 0; i < cv.Persone(); i++)
     cout << i << ": " << cv.VediPaziente(i)->Nome()
          << " " << cv.VediPaziente(i)->Cognome() << endl;
   cout << "Numero? ";
   cin >> num_persona;
   cout << "Vaccini disponibili : " << endl;
   for (i = 0; i < cv.Vaccini(); i++)
     cout << i << ": " << cv.VediVaccino(i)->Nome() << endl;
   cout << "Numero? ";
   cin >> num_vaccino;
   cout << "Data : ";
   cin >> data;
   p = cv.VediPaziente(num_persona);
   v = cv.VediVaccino(num_vaccino);
   cv.InserisciVaccinazione(v,p,data);
   break;
 case 5:
   cout << "Persone presenti : " << endl;
   for (i = 0; i < cv.Persone(); i++)
     cout << i << ": " << cv.VediPaziente(i)->Nome()
          << " " << cv.VediPaziente(i)->Cognome() << endl;
   cout << "Numero? ";
   cin >> num_persona;
   cout << "Malattie trattate : " << endl;
   for (i = 0; i < cv.Malattie(); i++)
     cout << i << ": " << cv.VediMalattia(i)->Codice() << endl;
   cout << "Numero? ";
   cin >> num_malattia;
   p = cv.VediPaziente(num_persona);
   m = cv.VediMalattia(num_malattia);
   m->InserisciMalato(p);
   break;
 case 6:
   cout << "Vaccini disponibili : " << endl;
   for (i = 0; i < cv.Vaccini(); i++)
```

```
            cout << i << ": " << cv.VediVaccino(i)->Nome() << endl;
          cout << "Numero? ";
          cin >> num_vaccino;
          cout << "Malattie trattate : " << endl;
          for (i = 0; i < cv.Malattie(); i++)
            cout << i << ": " << cv.VediMalattia(i)->Codice() << endl;
          cout << "Numero? ";
          cin >> num_malattia;
          v = cv.VediVaccino(num_vaccino);
          m = cv.VediMalattia(num_malattia);
          v->InserisciMalattia(m);
          break;
        case 7:
          {
            vector<Vaccino*> v;
            vector<Persona*> p;
            int importanza, num;
            cout << "Vaccini disponibili : " << endl;
            for (i = 0; i < cv.Vaccini(); i++)
              cout << i << ": " << cv.VediVaccino(i)->Nome() << endl;
            cout << "Inserisci i numeri (-1 per terminare) ";
            do
              {
                cin >> num;
                if (num != -1)
                  v.push_back(cv.VediVaccino(num));
              }
            while (num != -1);
            cout << "Importanza : ";
            cin >> importanza;
            cout << "Data : ";
            cin >> data;
            p = PersoneVaccinate(data,importanza,v);
            cout << "Persone vaccinate" << endl;
            for (i = 0; i < p.size(); i++)
              cout << *(p[i]) << endl;

          }
        }
    }
  while (scelta != 0);
}

vector<Persona*> PersoneVaccinate(Data d, unsigned n, const vector<Vaccino*>& v)
{
  vector<Persona*> persone;
  for (unsigned i = 0; i < v.size(); i++)
    {
      if (v[i]->Importanza() > n)
        for (unsigned j = 0; j < v[i]->Vaccinazioni(); j++)
          if (v[i]->VediVaccinazione(j).DataVaccinazione() < d)
            persone.push_back(v[i]->VediVaccinazione(j).Vaccinato());
    }
  return persone;
}
```

Soluzione compito del 2 dicembre 2003

Soluzione esercizio 1: File `Specie.h`

La responsabilità dell'associazione SiNutreDi viene assegnata da un solo lato, in particolare dal lato dell'oggetto predatore (l'altra scelta sarebbe stata equivalente).

```
// File Specie.h
#ifndef SPECIE_H
#define SPECIE_H

#include <vector>
#include <string>

using namespace std;

class Specie
{
  friend bool operator==(const Specie& s1, const Specie& s2);
public:
  Specie(string n);
  string Nome() const { return nome; }
  void AggiungiSiNutreDi(Specie *s);
  Specie* Nutrimento(unsigned i) const { return si_nutre_di[i]; }
  bool SiNutreDi(Specie* s) const;
private:
  string nome;
  vector<Specie*> si_nutre_di;
};
#endif
```

Soluzione esercizio 1: File `Animale.h` e `BioParco.h`

La responsabilità delle associazioni Tipo e `Risiede` è data solo alla classe `Animale`, mentre le due associazioni Appartiene sono di responsabilità della classe `BioParco`.

```
// File Animale.h
#ifndef ANIMALE_H
#define ANIMALE_H

#include "Gabbia.h"
#include "Specie.h"

class Animale
{
  friend ostream& operator<<(ostream& os, const Animale& a);
public:
  Animale(string n, unsigned e, Specie* s, Gabbia* g);
  Specie* QualeSpecie() const { return specie; }
  Gabbia* QualeGabbia() const { return gabbia; }
  string Nome() const { return nome; }
  unsigned Eta() const { return eta; }

  // Funzione non richiesta, inserita solo per la verifica dell'esercizio 4
  void CambiaEta(unsigned e) { eta = e; }
private:
  string nome;
  unsigned eta;
  Gabbia* gabbia;
  Specie* specie;
```

```
};
#endif

// File BioParco.h
#ifndef BIOPARCO_H
#define BIOPARCO_H

#include "Gabbia.h"
#include "Animale.h"

class BioParco
{
  friend ostream& operator<<(ostream& os, const BioParco& bp);
public:
  BioParco(string n, unsigned p);
  BioParco(const BioParco&);
  BioParco& operator=(const BioParco&);
  string Nome() const { return nome; }
  unsigned Prezzo() const { return prezzo; }
  unsigned NumAnimali() const { return animali.size(); }
  unsigned NumGabbie() const { return gabbie.size(); }
  Animale* VediAnimale(unsigned i) const { return animali[i]; }
  Gabbia* VediGabbia(unsigned i) const { return gabbie[i]; }
  bool InserisciAnimale(string n, unsigned e, Specie* s, Gabbia* g);
  bool InserisciGabbia(Gabbia* g);
private:
  int CercaGabbia(string n) const;
  int CercaAnimale(string n) const;
  string nome;
  unsigned prezzo;
  vector<Gabbia*> gabbie;
  vector<Animale*> animali;
};
#endif
```

Soluzione esercizio 2 e 4

La definizione delle funzioni delle classi non presentano particolarità di rilievo.

```
// File Specie.cpp
#include "Specie.h"

Specie::Specie(string n)
  : nome(n)
{}

void Specie::AggiungiSiNutreDi(Specie* s)
{
  for (unsigned i = 0; i < si_nutre_di.size(); i++)
    if (*(si_nutre_di[i]) == *s)
      return;  // in questo caso non inserisce (cfr. specifiche)
  si_nutre_di.push_back(s);
}

bool Specie::SiNutreDi(Specie* s) const
{
  for (unsigned i = 0; i < si_nutre_di.size(); i++)
    if (*(si_nutre_di[i]) == *s)
      return true;
  return false;
```

```
}

bool operator==(const Specie& s1, const Specie& s2)
{
  return s1.nome == s2.nome;
}

// File Animale.cpp
#include "Animale.h"

Animale::Animale(string n, unsigned e,  Specie* s, Gabbia* g)
  : nome(n)
{
  eta = e;
  specie = s;
  gabbia = g;
}

ostream& operator<<(ostream& os, const Animale& a)
{
  os << a.nome << " (" << a.specie->Nome() << " di " << a.eta << " anni) in " << a.gabbia->Nome();
  return os;
}

// File BioParco.cpp

#include "BioParco.h"

BioParco::BioParco(string n, unsigned p)
  : nome(n)
{
  prezzo = p;
}

BioParco::BioParco(const BioParco& bp)
  : nome(bp.nome), gabbie(bp.gabbie), animali(bp.animali.size())
{
  unsigned i;
  prezzo = bp.prezzo;
  for (i = 0; i < animali.size(); i++)
    animali[i] = new Animale(*(bp.animali[i]));
}

BioParco& BioParco::operator=(const BioParco& bp)
{
  unsigned i;
  nome = bp.nome;
  prezzo = bp.prezzo;
  gabbie = bp.gabbie;

  for (i = 0; i < animali.size(); i++)
    delete animali[i];
  animali.resize(bp.animali.size());
  for (i = 0; i < animali.size(); i++)
    animali[i] = new Animale(*(bp.animali[i]));
  return *this;
}

bool BioParco::InserisciAnimale(string n, unsigned e, Specie* s, Gabbia* g)
{
```

```
  if (CercaAnimale(n) != -1)
    return false;
  else if (CercaGabbia(g->Nome()) == -1)
    return false;
  else
    {
      animali.push_back(new Animale(n,e,s,g));
      return true;
    }
}

bool BioParco::InserisciGabbia(Gabbia* g)
{
  if (CercaGabbia(g->Nome()) != -1)
    return false;
  else
    {
      gabbie.push_back(g);
      return true;
    }
}

int BioParco::CercaGabbia(string n) const
{
  for (unsigned i = 0; i < gabbie.size(); i++)
    if (gabbie[i]->Nome() == n)
      return i;
  return -1;
}

int BioParco::CercaAnimale(string n) const
{
  for (unsigned i = 0; i < animali.size(); i++)
    if (animali[i]->Nome() == n)
      return i;
  return -1;
}

ostream& operator<<(ostream& os, const BioParco& bp)
{
  unsigned i;
  os << "Stato del BioParco " << bp.nome << ": " << endl;
  os << "Animali: " << endl;
  for (i = 0; i < bp.animali.size() ; i++)
    os << *(bp.animali[i]) << endl;
  os << "Gabbie: " << endl;
  for (i = 0; i < bp.gabbie.size() ; i++)
    os << *(bp.gabbie[i]) << endl;
  return os;
}
```

Soluzione esercizio 3 + driver

```
// File main.cpp

#include "BioParco.h"

unsigned ContaAnimaliInPericolo(const BioParco& bp);
```

```
int main()
{
  BioParco bp("Parco Verde",3);
  Specie s1("Leone"), s2("Zebra"), s3("Antilope");
  Gabbia g1("Parchetto",60), g2("Bosco",45);

  s1.AggiungiSiNutreDi(&s2);
  bp.InserisciGabbia(&g1);
  bp.InserisciGabbia(&g2);
  bp.InserisciAnimale("Bobo",3,&s1,&g1);
  bp.InserisciAnimale("Lillo",5,&s2,&g2);
  bp.InserisciAnimale("Pimpi",5,&s3,&g1);
  bp.InserisciAnimale("Silli",6,&s3,&g2);
  bp.InserisciAnimale("Pappi",6,&s3,&g1);

  // Verifica inserimenti
  cout << bp;

  // Test animali in pericolo
  cout << "Ci sono " << ContaAnimaliInPericolo(bp) << " animali in pericolo " << endl;
  s1.AggiungiSiNutreDi(&s3);
  cout << "Ci sono " << ContaAnimaliInPericolo(bp) << " animali in pericolo " << endl;

  // Test funzioni speciali: l'oggetto bp non viene modificato dalla modifica su bp2
  BioParco bp2 = bp;
  bp2.VediAnimale(0)->CambiaEta(6);
  cout << bp;

  return 0;
}

unsigned ContaAnimaliInPericolo(const BioParco& bp)
{
  unsigned i, j, conta = 0;

  for (i = 0; i < bp.NumAnimali(); i++)
    for (j = 0; j < bp.NumAnimali(); j++)
      {
        if (i != j && bp.VediAnimale(j)->QualeSpecie()->SiNutreDi(bp.VediAnimale(i)->QualeSpecie())
           && bp.VediAnimale(j)->QualeGabbia()->Nome() == bp.VediAnimale(i)->QualeGabbia()->Nome())
        {
          conta++;
          break;
        }
      }
  return conta;
}
```

Soluzione compito del 19 marzo 2004

Soluzione esercizio 1

La soluzione più semplice per questo esercizio consiste nell'usare una struttura uguale alla classe `Pila` fatta in classe. In questo caso gli inserimenti e le eliminazioni in testa si gestiscono come nella pila

normale, mentre per gli inserimenti e le eliminazioni in coda si deve traslare tutta la pila di una locazione in avanti nel vettore.

Nella soluzione qui proposta si è scelta una strada alternativa che esegue le modifiche in coda in modo più efficiente. La pila è memorizzata non nella parte bassa del vettore, ma nella parte centrale. Questo permette di inserire in entrambi i lati senza dover spostare elementi. Quando da una delle estremità non c'è più spazio, si alloca un vettore più grande e si riposizione la pila al centro del nuovo vettore.

Si noti che con questa scelta si rialloca il vettore anche quando non è strettamente necessario, in quanto sarebbe sufficiente traslare la pila al suo interno. Nella soluzione scelta però si minimizzano i casi in cui bisogna spostare la pila.

```
// File PilaBifronte.h
#ifndef PILABIFRONTE_H
#define PILABIFRONTE_H
#include <iostream>

using namespace std;

class PilaBifronte
{
  friend ostream& operator<<(ostream& os, const PilaBifronte& p);
  friend istream& operator>>(istream&, PilaBifronte& p);
public:
  PilaBifronte();
  PilaBifronte(const PilaBifronte& p);
  ~PilaBifronte();
  PilaBifronte& operator=(const PilaBifronte& p);
  void Push(int elem,bool in_testa);
  void Pop(bool in_testa);
  int Top(bool in_testa) const;
  bool EstVuota() const { return testa == - 1; }
 private:
  void IngrandisciERiposiziona();
  int* vet;     // vettore di interi
  int dim;      // dimensione del vettore
  int testa, coda; // elemento affiorante in testa e in coda
};
#endif
```

Soluzione esercizio 2

```
//File PilaBifronte.cpp
#include "PilaBifronte.h"

PilaBifronte::PilaBifronte()
{
  dim = 100;
  vet = new int[dim];
  testa = -1;
  coda = -1;
}

PilaBifronte::~PilaBifronte()
{
  delete [] vet;
}

PilaBifronte::PilaBifronte(const PilaBifronte& p)
{
  dim = p.dim;
  testa = p.testa;
```

```
  coda = p.coda;
  vet = new int[dim];
  if (testa != -1)
    for (int i = coda; i <= testa; i++)
      vet[i] = p.vet[i];
}

PilaBifronte& PilaBifronte::operator=(const PilaBifronte& p)
{
  if (p.testa >= dim)
    {
      delete [] vet;
      dim = p.dim;
      vet = new int[dim];
    }
  testa = p.testa;
  coda = p.coda;
  if (testa != -1)
    for (int i = coda; i <= testa; i++)
      vet[i] = p.vet[i];
  return *this;
}

int PilaBifronte::Top(bool in_testa) const
{
  assert(!EstVuota());
  if (in_testa)
    return vet[testa];
  else
    return vet[coda];
}

void PilaBifronte::Push(int elem, bool in_testa)
{
  if (testa == -1)
    { // indipendentemente da in_testa inserisco al centro del vettore
      unsigned centro = dim/2-1;
      vet[centro] = elem;
      testa = centro;
      coda = centro;
    }
  else
    {
      if (in_testa)
        {
          if (testa == dim - 1)
            IngrandisciERiposiziona();
          testa++;
          vet[testa] = elem;
        }
      else
        {
          if (coda == 0)
            IngrandisciERiposiziona();
          coda--;
          vet[coda] = elem;
        }
    }
}
```

```
void PilaBifronte::IngrandisciERiposiziona()
{
  // ingrandisci il vettore e riposiziona la pila al centro del vettore
  int* aux_vet = new int[dim*2];
  unsigned lunghezza = testa - coda + 1;
  unsigned new_coda = dim - lunghezza/2;
  unsigned j, i;
  for (i = coda, j = new_coda; i <= testa; i++,j++)
    aux_vet[j] = vet[i];
  delete [] vet;
  vet = aux_vet;
  dim = 2*dim;
  coda = new_coda;
  testa = coda + lunghezza - 1;
}

void PilaBifronte::Pop(bool in_testa)
{
  assert(!EstVuota());
  if (in_testa)
    testa--;
  else
    coda++;
  if (coda > testa)
    {// la pila e' vuota
      testa = -1;
      coda = -1;
    }
}

ostream& operator<<(ostream& os, const PilaBifronte& p)
{
  os << "(";
  if (p.testa != -1)
    {
      for (int i = p.coda; i < p.testa; i++)
        os << p.vet[i] << ", ";
      os << p.vet[p.testa];
    }
  os << ")";
  return os;
}

istream& operator>>(istream& is, PilaBifronte& p)
{
  // assume gli elementi della pila tra parentesi e separati da virgole
  int elem;
  char ch;

  p.testa = -1; // Svuota la pila

  is >> ch; // legge la parentesi aperta
  ch = is.peek();
  if (ch != ')')
    {
      do
        {
          is >> elem >> ch;
```

```
          p.Push(elem,true); // il problema del ridimensionamento e'
                             // demandato a Push()
        }
      while (ch != ')');
    }
  else
    is >> ch;
  return is;
}
```

Soluzione esercizi 3 + driver

```
#include <fstream>
#include "PilaBifronte.h"

void ModificaPila(PilaBifronte& pb1, const PilaBifronte& pb2);

int main()
{
  PilaBifronte p;
  unsigned scelta;
  bool in_testa;
  char scelta_in_testa;
  char nome_file[20];
  ifstream is;
  ofstream os;

  cout << "Nome del file contenete lo stato attuale della pila : ";
  cin >> nome_file;
  is.open(nome_file);
  is >> p;
  is.close();
  do
    {
      cout << "PilaBifronte : " << p << endl;
      cout << "Operazione: " << endl
           << "  1. Push" << endl
           << "  2. Pop" << endl
           << "  3. Top" << endl
           << "  4. Test Funzione Esterna" << endl
           << "  0. Esci" << endl
           << "Scelta : ";
      cin >> scelta;

      switch(scelta)
        {
        case 1:
          {
            int elem;
            cout << "Elemento : ";
            cin >> elem;
            cout << "Lato (t/c) : ";
            cin >> scelta_in_testa;
            if (scelta_in_testa == 'T' || scelta_in_testa == 't')
              in_testa = true;
            else
              in_testa = false;
            p.Push(elem, in_testa);
            break;
```

```
          }
        case 2:
          cout << "Lato (t/c) : ";
          cin >> scelta_in_testa;
          if (scelta_in_testa == 'T' || scelta_in_testa == 't')
            in_testa = true;
          else
            in_testa = false;
          p.Pop(in_testa);
          break;
        case 3:
          cout << "Lato (t/c) : ";
          cin >> scelta_in_testa;
          if (scelta_in_testa == 'T' || scelta_in_testa == 't')
            in_testa = true;
          else
            in_testa = false;
          cout << "Elemento affiorante: " << p.Top(in_testa) << endl;
        case 4:
          {
            PilaBifronte p2 = p;
            ModificaPila(p,p2);
            break;
          }
        case 0:
          break;
        default:
          cout << "Scelta non valida" << endl;
        }
    }
  while (scelta != 0);
  os.open(nome_file);
  os << p;
}

void ModificaPila(PilaBifronte& pb1, const PilaBifronte& pb2)
{
  if (pb2.EstVuota())
    pb1.Push(0,true);
  else
    {
      if (pb2.Top(true) < pb2.Top(false))
        pb1.Push(pb2.Top(true),true);
      else
        pb1.Push(pb2.Top(false),true);
    }
}
```

Soluzione compito del 28 giugno 2004

Soluzione esercizio 1

Non potendo usare (per specifica) la classe template `pair` della STL, si è deciso di realizzare la classe `Elemento` che crea una coppia *ad hoc*. La classe `VettoreCompatto` utilizza un vettore, chiamato `vet`, di `Elemento`.

```
// file VettoreCompatto.h
#ifndef VETTORE_COMPATTO_H
#define VETTORE_COMPATTO_H

#include <iostream>

using namespace std;

class Elemento
{
 public:
  Elemento(int v = 0, unsigned r = 1) { valore = v; ripetizioni = r; }
  int valore;
  unsigned ripetizioni;
};

class VettoreCompatto
{
  friend ostream& operator<<(ostream& os, const VettoreCompatto& vc);
 public:
  VettoreCompatto();
  VettoreCompatto(const VettoreCompatto&);
  ~VettoreCompatto();
  VettoreCompatto& operator=(const VettoreCompatto&);
  void Inserisci(int e);
  void Elimina();
  int operator[](unsigned i) const;
  unsigned NumElem() const { return num_elem; }
 private:
  Elemento* vet;
  unsigned num_elem; // numero totale di elementi nel VettoreCompatto
  unsigned dim; // dimensione di vet
};

#endif
```

Soluzione esercizio 2

Si è scelto di ridimensionare il vettore ad ogni singolo inserimento o cancellazione nel vettore `vet`. Una scelta alternativa, altrettanto valida, eè quella di mantenere un vettore sovradimensionato (cfr. classe `Pila` spiegata in classe).

Non tutte le operazioni di inserimento provocano un ridimensionamento del vettore, ma solo quelle per cui l'elemento inserito è diverso dall'ultimo del vettore.

L'operatore `[]`, seguendo le specifiche, viene realizzato solo nella forma di selettore, non per modificare il vettore. La selezione avviene contando gli elementi dal primo, ciascuno con la sua molteplicità.

```
// file VettoreCompatto.cpp
#include "VettoreCompatto.h"

VettoreCompatto::VettoreCompatto()
{
  vet = NULL;
  num_elem = 0;
  dim = 0;
}

VettoreCompatto::VettoreCompatto(const VettoreCompatto& v)
{
  dim = v.dim;
  num_elem = v.num_elem;
```

```
  vet = new Elemento[dim];
  for (unsigned i = 0; i < dim; i++)
    vet[i] = v.vet[i];
}

VettoreCompatto::~VettoreCompatto()
{
  delete [] vet;
}

VettoreCompatto& VettoreCompatto::operator=(const VettoreCompatto& v)
{
  delete [] vet;
  dim = v.dim;
  num_elem = v.num_elem;
  vet = new Elemento[dim];
  for (unsigned i = 0; i < dim; i++)
    vet[i] = v.vet[i];
  return *this;
}

void VettoreCompatto::Inserisci(int e)
{
  if (num_elem == 0)
    {
      vet = new Elemento(e,1);
      dim = 1;
    }
  else if (vet[dim-1].valore == e)
    vet[dim-1].ripetizioni++;
  else
    {
      Elemento* aux_vet = new Elemento[dim+1];
      for (unsigned i = 0; i < dim; i++)
        aux_vet[i] = vet[i];
      aux_vet[dim] = Elemento(e,1);
      delete [] vet;
      vet = aux_vet;
      dim++;
    }
  num_elem++;
}

void VettoreCompatto::Elimina()
{
  assert (num_elem > 0);
  if (vet[dim-1].ripetizioni > 1)
    vet[dim-1].ripetizioni--;
  else if (num_elem == 1)
    {
      delete vet;
      vet = NULL;
      dim = 0;
    }
  else
    {
      Elemento* aux_vet = new Elemento[dim-1];
      for (unsigned i = 0; i < dim-1; i++)
        aux_vet[i] = vet[i];
```

```
      delete [] vet;
      vet = aux_vet;
      dim--;
    }
  num_elem--;
}

int VettoreCompatto::operator[](unsigned i) const
{
  unsigned k = 0, h = 0;
  for (unsigned j = 0; j < i; j++)
    {
      h++;
      if (h == vet[k].ripetizioni)
        {
          h = 0;
          k++;
        }
    }
  return vet[k].valore;
}

ostream& operator<<(ostream& os, const VettoreCompatto& vc)
{
  os << "[";
  if (vc.dim > 0)
    {
      for (unsigned i = 0; i < vc.dim; i++)
        os << "(" << vc.vet[i].valore << "," << vc.vet[i].ripetizioni << ")";
    }
  os << "]";
  return os;
}
```

Soluzione esercizi 3 + driver

Disponendo dell'operatore `[]` la funzione `Inverti` risulta particolarmente semplice.

Per il *driver*, per brevità si è proceduto solo ad alcuni inserimenti mirati, ed alle successive eliminazioni di tutti gli elementi (per verificare il comportamento nel caso di vettore vuoto).

```
// file main.cpp
#include "VettoreCompatto.h"

VettoreCompatto Inverso(const VettoreCompatto& v);

int main()
{
  const int n = 20;
  int i;
  VettoreCompatto v;

  for (i = 0; i < n; i++)
    v.Inserisci(i/3);

  // Stampa il vettore esteso
  for (i = 0; i < n; i++)
    cout << v[i] << " ";
  // Stampa il vettore compatto
  cout << endl;
  cout << v << endl;
```

```
  cout << Inverso(v) << endl;


  for (i = 0; i < n; i++)
    {
      v.Elimina();
      cout << v << endl;
    }

  for (i = 0; i < n; i++)
    v.Inserisci(-i/8);

  cout << v << endl;
}

VettoreCompatto Inverso(const VettoreCompatto& v)
{
  VettoreCompatto ris;
  for (int i = v.NumElem() - 1; i >= 0; i--)
    ris.Inserisci(v[i]);
  return ris;
}
```

Soluzione compito del 6 dicembre 2004

Soluzione esercizio 1

```
// yacul.h
#ifndef YACUL_H
#define YACUL_H

#include <iostream>
#include <string>
#include <vector>

using namespace std;

class Dichiarazione
{
  friend istream& operator>>(istream& is, Dichiarazione& d);
  friend ostream& operator<<(ostream& os, const Dichiarazione& d);
 public:
  Dichiarazione() {}
  Dichiarazione(string n, int v) : nome(n) { valore = v; }
  string Nome() const { return nome; }
  int Valore() const { return valore; }
  void SetValore(int v) { valore = v; }
 private:
  string nome;
  int valore;
};

class Istruzione
{
  friend istream& operator>>(istream& is,
                             Istruzione& ist);
  friend ostream& operator<<(ostream& os,
```

```
                              const Istruzione& ist);
 public:
  Istruzione() {}
  Istruzione(string o1, string o2, char op, string r)
    : operando1(o1), operando2(o2), risultato(r)
    { operatore = op; }
  char Operatore() const { return operatore; }
  string Operando1() const { return operando1; }
  string Operando2() const { return operando2; }
  string Risultato() const { return risultato; }
 private:
  char operatore;
  string operando1, operando2, risultato;
};

class Yatul
{
  friend istream& operator>>(istream& is, Yatul& ist);
  friend ostream& operator<<(ostream& os, const Yatul& ist);
 public:
  Yatul();
  Dichiarazione* VediDichiarazione(unsigned i) const { return dichiarazioni[i]; }
  Istruzione* VediIstruzione(unsigned i) const { return istruzioni[i]; }
  unsigned NumDichiarazioni() const { return dichiarazioni.size(); }
  unsigned NumIstruzioni() const { return istruzioni.size(); }
  void InserisciDichiarazione(Dichiarazione* d);
  void InserisciIstruzione(Istruzione* i);
  bool ProgrammaCorretto() const;
  int CalcolaValore(string nome) const;
 private:
  vector<Istruzione*> istruzioni;
  vector<Dichiarazione*> dichiarazioni;
  int CercaVariabile(string nome) const;
};

#endif
```

Soluzione esercizio 2, 3, e 4

Si noti che la funzione `CalcolaValore` utilizza un vettore di interi che memorizza i valori finali delle variabili. Sarebbe errato invece modificare i valori contenuti nelle dichiarazioni, perché questo comporterebbe un risultato sbagliato quando la funzione viene invocata più volte.

```
// yacul.cpp
#include "yatul.h"
#include <cassert>

istream& operator>>(istream& is, Dichiarazione& d)
{
  char ch;
  is >> d.nome >> ch >> d.valore;
  return is;
}

ostream& operator<<(ostream& os, const Dichiarazione& d)
{
  os << d.nome << " = " << d.valore;
  return os;
}
```

```
istream& operator>>(istream& is, Istruzione& i)
{
  string s;
  is >> i.risultato >> s >> i.operando1 >> i.operatore >> i.operando2;
  assert(i.operatore == '+' || i.operatore == '-');
  return is;
}

ostream& operator<<(ostream& os, const Istruzione& i)
{
  os << i.risultato << " <-- " << i.operando1 << " " << i.operatore << " " << i.operando2;
  return os;
}

Yatul::Yatul()
  :  dichiarazioni(1)
{
  dichiarazioni[0] = new Dichiarazione("alfa",0);
}

void Yatul::InserisciDichiarazione(Dichiarazione* d)
{
  assert(CercaVariabile(d->Nome()) == -1);
  dichiarazioni.push_back(d);
}

void Yatul::InserisciIstruzione(Istruzione* i)
{
  assert(i->Operatore() == '+' || i->Operatore() == '-');
  istruzioni.push_back(i);
}

bool Yatul::ProgrammaCorretto() const
{
  unsigned i;
  for (i = 0; i < istruzioni.size(); i++)
    if (CercaVariabile(istruzioni[i]->Operando1()) == -1
        || CercaVariabile(istruzioni[i]->Operando2()) == -1
        || CercaVariabile(istruzioni[i]->Risultato()) == -1)
      return false;
  return true;
}

int Yatul::CalcolaValore(string nome) const
{
  unsigned i, r, o1, o2;
  vector<int> valore_corrente(dichiarazioni.size());

  assert(CercaVariabile(nome) != -1);
  assert(ProgrammaCorretto());

  for (i = 0; i < dichiarazioni.size(); i++)
    valore_corrente[i] = dichiarazioni[i]->Valore();

  for (i = 0; i < istruzioni.size(); i++)
    {
      r = CercaVariabile(istruzioni[i]->Risultato());
      o1 = CercaVariabile(istruzioni[i]->Operando1());
      o2 = CercaVariabile(istruzioni[i]->Operando2());
```

```
      if (istruzioni[i]->Operatore() == '+')
        valore_corrente[r] = valore_corrente[o1] + valore_corrente[o2];
      else
        valore_corrente[r] = valore_corrente[o1] - valore_corrente[o2];
    }
  return valore_corrente[CercaVariabile(nome)];
}

int Yatul::CercaVariabile(string nome) const
{
  unsigned i;
  for (i = 0; i < dichiarazioni.size(); i++)
    if (dichiarazioni[i]->Nome() == nome)
      return i;
  return -1;
}

istream& operator>>(istream& is, Yatul& yp)
{
  string s;
  char ch;
  unsigned i;
  unsigned num_dichiarazioni, num_istruzioni;

  is >> s >> num_dichiarazioni >> ch >> num_istruzioni;

  yp.dichiarazioni.resize(num_dichiarazioni);
  yp.istruzioni.resize(num_istruzioni);

  for (i = 0; i < num_dichiarazioni; i++)
    {
      yp.dichiarazioni[i] = new Dichiarazione;
      is >>  *(yp.dichiarazioni[i]);
    }
  for (i = 0; i < num_istruzioni; i++)
    {
      yp.istruzioni[i] = new Istruzione;
      is >>  *(yp.istruzioni[i]);
    }
  return is;
}

ostream& operator<<(ostream& os, const Yatul& yp)
{
  unsigned i;
  os << "#yatul " << yp.dichiarazioni.size() << '/' << yp.istruzioni.size() << endl;
  for (i = 0; i < yp.dichiarazioni.size(); i++)
    os << *(yp.dichiarazioni[i]) << endl;
  for (i = 0; i < yp.istruzioni.size(); i++)
    os << *(yp.istruzioni[i]) << endl;
  return os;
}
```

Soluzione esercizi 5 + driver

```
// main.cpp
#include <fstream>
#include "yatul.h"
```

```
bool VariabileInutile(const Yatul& p);

int main()
{
  string nome_file, var;
  ifstream is("prova.ytl");
  Dichiarazione d("beta",10);
  Istruzione i("alfa","beta",'+',"alfa");
  Yatul p;

  cout << "Programma generato dal costruttore" << endl;
  cout << p << endl;

  p.InserisciDichiarazione(&d);
  p.InserisciIstruzione(&i);

  cout << "Programma costruito con gli inserimenti" << endl;
  cout << p << endl;

  is >> p;
  cout << "Programma letto dal file" << endl;
  cout << p << endl;

  if (p.ProgrammaCorretto())
    {
      cout << "Il programma e' sintatticamente corretto" << endl;
      do
        {
          cout << "Inserisci il nome di una variabile (fine per uscire): ";
          cin >> var;
          if (var != "fine")
            cout << "Il valore calcolato e' : " << p.CalcolaValore(var) << endl;
        }
      while (var != "fine");
    }
  else
    cout << "Ci sono errori sintattici" << endl;

  if (VariabileInutile(p))
    cout << "Ci sono variabili inutili" << endl;
  else
    cout << "Non ci sono variabili inutili" << endl;
}

bool VariabileInutile(const Yatul& p)
{
  unsigned i,j;
  bool usata;

  for (i = 0; i < p.NumDichiarazioni(); i++)
    {
      usata = false;
      for (j = 0; j < p.NumIstruzioni(); j++)
        {
          if (p.VediIstruzione(j)->Operando1() == p.VediDichiarazione(i)->Nome()
              || p.VediIstruzione(j)->Operando2() == p.VediDichiarazione(i)->Nome())
             usata = true;
        }
      if (!usata)
```

```
        return true;
    }
  return false;
}
```

Soluzione compito del 20 dicembre 2004

Soluzione esercizio 1

Il problema principale di questo compito era quello di decidere come rappresentare i dati della scacchiera. La soluzione che appare più semplice è quelle di memorizzare unicamente le caselle contenenti gli oggetti: un vettore per le bombe. un singolo valore per il tesoro e il giocatore. La memorizzazione della scacchiera come matrice esplicita risulta più complessa e macchinosa.

```
// file yaig.h
#ifndef YAIG_H
#define YAIG_H
#include <iostream>
#include <vector>
#include "casella.h"

using namespace std;

class Yaig
{
  friend ostream& operator<<(ostream& os, const Yaig& u);
 public:
  Yaig(unsigned dim, unsigned num_b);
  unsigned Muovi(char dir);
  unsigned Esplora(unsigned r);
  Casella Posizione() const { return posizione; }
  vector<Casella> Bombe() const { return bombe; }
  unsigned ScoppiRimasti() const { return max_scoppi - scoppi; }
  unsigned Dim() const { return dim; }
 private:
  unsigned dim;
  unsigned scoppi, max_scoppi;
  Casella posizione, tesoro;
  vector<Casella> bombe;
  bool CasellaProibita(Casella c);
  bool EstBomba(Casella c);
};
#endif
```

Soluzione esercizio 2

```
// File yaig.cpp
#include "yaig.h"
#include "../Utilities/Random.h"
#include <cassert>

Yaig::Yaig(unsigned d, unsigned num_b)
  :  bombe(num_b)
{
  Casella c;
  unsigned i;
```

```
  dim = d;
  max_scoppi = num_b/2; // arrotondato per difetto
  scoppi = 0;
  assert (num_b < dim * dim / 2);

  for (i = 0; i < num_b; i++)
    {
      do
        c.Set(Random(0,dim-1), Random(0,dim-1));
      while (CasellaProibita(c));
      bombe[i] = c;
    }
  do
    tesoro.Set(Random(0,dim-1), Random(0,dim-1));
  while (CasellaProibita(tesoro));
}

bool Yaig::CasellaProibita(Casella c)
{
  return (c.Riga() == 0 && c.Colonna() == 0)
    || EstBomba(c);
}

bool Yaig::EstBomba(Casella c)
{
  for (unsigned i = 0; i < bombe.size(); i++)
    if (c == bombe[i])
      return true;
  return false;
}

unsigned Yaig::Muovi(char dir)
{
  Casella c = posizione;
  if (toupper(dir) == 'N')
    {
      if (posizione.Riga() < dim-1)
        c.Set(posizione.Riga()+1, posizione.Colonna());
    }
  else if (toupper(dir) == 'S')
    {
      if (posizione.Riga() > 0)
        c.Set(posizione.Riga()-1, posizione.Colonna());
    }
  else if (toupper(dir) == 'E')
    {
      if (posizione.Colonna() < dim-1)
        c.Set(posizione.Riga(), posizione.Colonna()+1);
    }
  else if (toupper(dir) == 'O')
    {
      if (posizione.Colonna() > 0)
        c.Set(posizione.Riga(), posizione.Colonna()-1);
    }
  else
    assert(false);

  if (c == tesoro)
    return 3;
```

```
  else if (EstBomba(c))
    if (scoppi < max_scoppi - 1)
      {
        scoppi++;
        return 1;
      }
    else
      return 2;
  else
    {
      posizione = c;
      return 0;
    }
}

unsigned Yaig::Esplora(unsigned r)
{
  unsigned conta = 0;
  Casella c;

  for (int i = posizione.Riga() - r; i <= int(posizione.Riga() + r); i++)
    for (int j = posizione.Colonna() - r; j <= int(posizione.Colonna() + r); j++)
      {
        if (i >= 0 && i < int(dim) && j >= 0 && j < int(dim))
          {
            c.Set(i,j);
            if (EstBomba(c))
              conta++;
          }
      }
  return conta;
}

ostream& operator<<(ostream& os, const Yaig& u)
{
  os << "Dimensione : " << u.dim << endl;
  os << "Posizione attuale: " << u.posizione << endl;
  os << "Tesoro : " << u.tesoro << endl;
  os << "Scoppi : " << u.scoppi << '/' << u.max_scoppi << endl;
  for (unsigned i = 0; i < u.bombe.size(); i++)
    os << "Bomba " << i << " in " << u.bombe[i] << endl;
  return os;
}
```

Soluzione esercizi 3

```
// File main.cpp
#include "yaig.h"

using namespace std;

int main()
{
  unsigned dimensione, bombe, mosse = 0, scelta, stato;
  cout << "Dimensione della scacchiera : ";
  cin >> dimensione;
  cout << "Numero di bombe : ";
  cin >> bombe;
  Yaig yaig(dimensione,bombe);
```

```
  do
    {
      //      cout << yaig; //   da inserire solo per il debugging
      cout << "Sei in posizione " << yaig.Posizione() << endl;
      cout << "Hai fatto " << mosse << " mosse" << endl;
      cout << "Hai ancora " << yaig.ScoppiRimasti() << " scoppi" << endl;
      cout << "Cosa vuoi fare (1: esplora, 2: muovi, 0: esci) : ";
      cin >> scelta;
      if (scelta == 1)
        {
          int r;
          cout << "Raggio di esplorazione : ";
          cin >> r;
          cout << "Ci sono " << yaig.Esplora(r) << " bombe nel raggio " << r << endl;
        }
      else if (scelta == 2)
        {
          char dir;
          cout << "Direzione (N,S,E,O) : ";
          cin >> dir;
          stato = yaig.Muovi(dir);
          if (stato == 1 || stato == 2)
            cout << "BUUUUUM" << endl;
          if (stato == 2 || stato == 3)
            break;
        }
      mosse++;
    }
  while (scelta != 0);
  cout << yaig << endl;
  if (stato == 2)
    cout << "HAI PERSO!!" << endl;
  else if (stato == 3)
    cout << "HAI VINTO!!" << endl;
  else
    cout << "CIAO, CIAO " << endl;
}
```

Soluzione compito del 5 dicembre 2005

Soluzione esercizio 1

In base alle operazioni delle classi, si è deciso di dare la responsabilità della relazione Mutuazione alla classe CorsoMutuato. Grazie alle molteplicità, questa scelta permette di gestire l'associazione con un unico puntatore. Per le associazioni con la classe `SessioneEsami`, la responsabilità è sempre solo di quest'ultima classe.

```
// File Corso.h
#ifndef CORSO_H
#define CORSO_H

#include "Esame.h"

using namespace std;

class Corso
```

```
{
  friend ostream& operator<<(ostream& os, const Corso& es);
  friend istream& operator>>(istream& is, Corso& es);
 public:
  Corso();
  Corso(const Corso& c);
  Corso(string n, string cs, string d, unsigned q);
  ~Corso();
  Corso& operator=(const Corso& c);
  string Nome() const { return nome; }
  string CorsoDiStudi() const { return corso_di_studi; }
  string Docente() const { return docente; }
  unsigned Quadrimestre() const { return quadrimestre; }
  unsigned NumAppelli() const;
  Esame* PrimoAppello() const { return primo_appello; }
  Esame* SecondoAppello() const { return secondo_appello; }
  void InserisciAppello(Esame* e);
 private:
  string nome;
  string corso_di_studi;
  string docente;
  unsigned quadrimestre;
  Esame* primo_appello;
  Esame* secondo_appello;
};

#endif
```

```
// File CorsoMutuato.h
#ifndef CORSO_MUTUATO_H
#define CORSO_MUTUATO_H

#include "Corso.h"

using namespace std;

class CorsoMutuato
{
  friend ostream& operator<<(ostream& os, const CorsoMutuato& es);
  friend istream& operator>>(istream& is, CorsoMutuato& es);
 public:
  CorsoMutuato();
  CorsoMutuato(string n, string cs, Corso* m);
  string Nome() const { return nome; }
  string CorsoDiStudi() const { return corso_di_studi; }
  Corso* Mutuante() const { return mutuante; }
  void InserisciMutuante(Corso* c);
 private:
  string nome;
  string corso_di_studi;
  Corso* mutuante;
};

#endif
```

```
// File SessioneEsami.h
#ifndef SESSIONE_ESAMI_H
#define SESSIONE_ESAMI_H

#include <vector>
```

```
#include "Esame.h"
#include "Corso.h"
#include "CorsoMutuato.h"

using namespace std;

class SessioneEsami
{
  friend ostream& operator<<(ostream& os, const SessioneEsami& es);
  friend istream& operator>>(istream& is, SessioneEsami& es);
 public:
  SessioneEsami(unsigned q);
  SessioneEsami(const SessioneEsami& se);
  ~SessioneEsami();
  SessioneEsami& operator=(const SessioneEsami& se);
  unsigned Quadrimestre() const { return quadrimestre; }
  unsigned Corsi() const { return corsi.size(); }
  unsigned CorsiMutuati() const { return corsi_mutuati.size(); }
  Corso* VediCorso(unsigned i) const { return corsi[i]; }
  CorsoMutuato* VediCorsoMutuato(unsigned i) const { return corsi_mutuati[i]; }
  void InserisciCorso(string n, string cs, string d, unsigned q);
  void EliminaCorso(string n, string cs);
  void InserisciCorsoMutuato(string n, string cs, string nm, string csm);
  void EliminaCorsoMutuato(string n, string cs);
  void InserisciAppelloCorso(string n, string cs, Data d, string a);
  vector<string> AuleOccupate(Data d);
 private:
  unsigned quadrimestre;
  vector<Corso*> corsi;
  vector<CorsoMutuato*> corsi_mutuati;
  int CercaCorso(string nome, string cs);
  int CercaCorsoMutuato(string nome, string cs);
  void DeallocaVettori(); // usato dal distruttore e dall'operatore <<
};

#endif
```

Soluzione esercizio 2, 4 e 5

Come evidenziati anche nel testo, tutti gli oggetti associati ad un oggetto della classe `SessioneEsami` sono allocati dinamicamente all'interno e non condivisi con l'esterno. Questo comporta la necessità di scrivere le tre funzioni speciali (costruttore di copia, operatore di assegnazione e distruttore) per la classe. Per gestire correttamente anche l'allocazione degli oggetti della classe `Esame`, non direttamente collegati alla classe `SessioneEsami`, è necessario dotare anche la classe `Corso` delle funzioni speciali.

```
// File Corso.cpp

#include "Corso.h"

Corso::Corso()
{
  primo_appello = NULL;
  secondo_appello = NULL;
}

Corso::Corso(const Corso& c)
  :  nome(c.nome), corso_di_studi(c.corso_di_studi), docente(c.docente)
{
  quadrimestre = c.quadrimestre;
  if (c.primo_appello != NULL)
```

```
    primo_appello = new Esame(*c.primo_appello);
  else
    primo_appello = NULL;
  if (c.secondo_appello != NULL)
    secondo_appello = new Esame(*c.secondo_appello);
  else
    secondo_appello = NULL;
}

Corso::Corso(string n, string cs, string d, unsigned q)
  :  nome(n), corso_di_studi(cs), docente(d)
{
  quadrimestre = q;
  primo_appello = NULL;
  secondo_appello = NULL;
}

Corso::~Corso()
{
  delete primo_appello;
  delete secondo_appello;
}

Corso& Corso::operator=(const Corso& c)
{
  nome = c.nome;
  corso_di_studi = c.corso_di_studi;
  docente = c.docente;
  quadrimestre = c.quadrimestre;

  if (primo_appello != NULL)
    delete primo_appello;
  if (secondo_appello != NULL)
    delete secondo_appello;


  if (c.primo_appello != NULL)
    primo_appello = new Esame(*c.primo_appello);
  else
    primo_appello = NULL;
  if (c.secondo_appello != NULL)
    secondo_appello = new Esame(*c.secondo_appello);
  else
    secondo_appello = NULL;
  return *this;
}

unsigned Corso::NumAppelli() const
{
  if (primo_appello == NULL)
    return 0;
  else if (secondo_appello == NULL)
    return 1;
  else
    return 2;
}

void Corso::InserisciAppello(Esame* e)
{
```

```
  if (primo_appello == NULL)
    primo_appello = e;
  else if (secondo_appello == NULL)
    {
      assert(e->DataProva() != primo_appello->DataProva());
      if (e->DataProva() > primo_appello->DataProva())
        secondo_appello = e;
      else
        {
          secondo_appello = primo_appello;
          primo_appello = e;
        }
    }
  else
    assert(false);
}

ostream& operator<<(ostream& os, const Corso& es)
{
  os << es.nome << ' ' << es.corso_di_studi << ' '
     << es.docente << ' ' << es.quadrimestre;
  return os;
}

istream& operator>>(istream& is, Corso& es)
{ // si assume che tutti i dati alfabetici siano privi di spazi
    is >> es.nome >> es.corso_di_studi >> es.docente
       >> es.quadrimestre;
  return is;
}
```

```
// File CorsoMutuato.cpp

#include "CorsoMutuato.h"

ostream& operator<<(ostream& os, const CorsoMutuato& es)
{
  os << es.nome << ' ' << es.corso_di_studi;
  return os;

}

istream& operator>>(istream& is, CorsoMutuato& es)
{// si assume che tutti i dati alfabetici siano privi di spazi
    is >> es.nome >> es.corso_di_studi;
  return is;

}

CorsoMutuato::CorsoMutuato()
{
  mutuante = NULL;
}

CorsoMutuato::CorsoMutuato(string n, string cs, Corso* m)
  : nome(n), corso_di_studi(cs)
{
  mutuante = m;
}
```

```
void CorsoMutuato::InserisciMutuante(Corso* c)
{
  mutuante = c;
}
```

```
// File SessioneEsami.cpp

#include "SessioneEsami.h"

SessioneEsami::SessioneEsami(unsigned q)
{
  quadrimestre = q;
}

SessioneEsami::SessioneEsami(const SessioneEsami& se)
  :  corsi(se.corsi.size()), corsi_mutuati(se.corsi_mutuati.size())
{
  unsigned i, j;
  for (i = 0; i < corsi.size(); i++)
    // Nota: la copia degli esami e' delegata al costruttore di copia di Corso
    corsi[i] = new Corso(*(se.corsi[i]));
  for (i = 0; i < corsi_mutuati.size(); i++)
    {
      corsi_mutuati[i] = new CorsoMutuato(*(se.corsi_mutuati[i]));
      j = CercaCorso(corsi_mutuati[i]->Mutuante()->Nome(), corsi_mutuati[i]->Mutuante()->CorsoDiStud
      corsi_mutuati[i]->InserisciMutuante(corsi[j]);
    }
}

SessioneEsami::~SessioneEsami()
{
  DeallocaVettori();
}

SessioneEsami& SessioneEsami::operator=(const SessioneEsami& se)
{
  unsigned i, j;

  DeallocaVettori();
  corsi.resize(se.corsi.size());
  corsi_mutuati.resize(se.corsi_mutuati.size());
  for (i = 0; i < corsi.size(); i++)
    corsi[i] = new Corso(*(se.corsi[i]));
  for (i = 0; i < corsi.size(); i++)
    {
      corsi_mutuati[i] = new CorsoMutuato(*(se.corsi_mutuati[i]));
      j = CercaCorso(corsi_mutuati[i]->Mutuante()->Nome(), corsi_mutuati[i]->Mutuante()->CorsoDiStud
      corsi_mutuati[i]->InserisciMutuante(corsi[j]);
    }
  return *this;
}

void SessioneEsami::DeallocaVettori()
{
  unsigned i;
  for (i = 0; i < corsi_mutuati.size(); i++)
    delete corsi_mutuati[i];
  for (i = 0; i < corsi.size(); i++)
    delete corsi[i];
  // Nota: gli esami vengono deallocati dal distruttore della classe Corso
```

```
}

int SessioneEsami::CercaCorso(string nome, string cs)
{
  for (unsigned i = 0; i < corsi.size(); i++)
    if (corsi[i]->Nome() == nome && corsi[i]->CorsoDiStudi() == cs)
      return i;
  return -1;
}

int SessioneEsami::CercaCorsoMutuato(string nome, string cs)
{

  for (unsigned i = 0; i < corsi_mutuati.size(); i++)
    if (corsi_mutuati[i]->Nome() == nome && corsi_mutuati[i]->CorsoDiStudi() == cs)
      return i;
  return -1;
}

void SessioneEsami::InserisciCorso(string n, string cs, string d, unsigned q)
{
  assert(CercaCorso(n,cs) == -1);
  assert(CercaCorsoMutuato(n,cs) == -1);

  corsi.push_back(new Corso(n,cs,d,q));
}

void SessioneEsami::EliminaCorso(string n, string cs)
{
  int i = CercaCorso(n,cs);
  unsigned j = 0;

  assert(i != -1);
  while(j < corsi_mutuati.size())
    if (corsi_mutuati[j]->Mutuante()->Nome() == n && corsi_mutuati[j]->Mutuante()->CorsoDiStudi() ==
      {
        delete corsi_mutuati[j];
        corsi_mutuati.erase(corsi_mutuati.begin() + j);
      }
    else
      j++;
  delete corsi[i];
  corsi.erase(corsi.begin() + i);
}

void SessioneEsami::InserisciCorsoMutuato(string n, string cs, string nm, string csm)
{
  int i = CercaCorso(nm,csm);
  assert (i != -1);
  assert(CercaCorsoMutuato(n,cs) == -1);
  assert(CercaCorso(n,cs) == -1);

  corsi_mutuati.push_back(new CorsoMutuato(n,cs,corsi[i]));
}

void SessioneEsami::EliminaCorsoMutuato(string n, string cs)
{
  int i = CercaCorsoMutuato(n,cs);
  assert(i != -1);
```

```
  corsi_mutuati.erase(corsi_mutuati.begin() + i);
}

void SessioneEsami::InserisciAppelloCorso(string n, string cs, Data d, string a)
{
  int i = CercaCorso(n,cs);
  assert (i != -1);
  corsi[i]->InserisciAppello(new Esame(d,a));
}

vector<string> SessioneEsami::AuleOccupate(Data d)
{
  vector<string> aule;
  for (unsigned i = 0; i < corsi.size(); i++)
    {
      if (corsi[i]->NumAppelli() >= 1)
        {
          if (corsi[i]->PrimoAppello()->DataProva() == d)
            {
              aule.push_back(corsi[i]->PrimoAppello()->Aula());
              continue;
            }
          if (corsi[i]->NumAppelli() == 2)
            if (corsi[i]->SecondoAppello()->DataProva() == d)
              {
                aule.push_back(corsi[i]->SecondoAppello()->Aula());
              }
        }
    }
  return aule;
}

ostream& operator<<(ostream& os, const SessioneEsami& es)
{
  unsigned i, esami = 0;
  os << "Quadrimestre: " << es.quadrimestre << endl;
  os << "Corsi: " << es.corsi.size() << endl;
  for (i = 0; i < es.corsi.size(); i++)
    {
      os << *(es.corsi[i]) << endl;
      esami += es.corsi[i]->NumAppelli();
    }

  os << "CorsiMutuati: " << es.corsi_mutuati.size() << endl;
  for (i = 0; i < es.corsi_mutuati.size(); i++)
    os << *(es.corsi_mutuati[i]) << ' ' << es.corsi_mutuati[i]->Mutuante()->Nome() << ' '
       << es.corsi_mutuati[i]->Mutuante()->CorsoDiStudi() << endl;

  os << "Esami: " << esami << endl;

   for (i = 0; i < es.corsi.size(); i++)
     if (es.corsi[i]->NumAppelli() >= 1)
       {
         os << es.corsi[i]->Nome() << ' ' << es.corsi[i]->CorsoDiStudi() << ' '
            << es.corsi[i]->PrimoAppello()->DataProva() << ' ' << es.corsi[i]->PrimoAppello()->Aula(
         if (es.corsi[i]->NumAppelli() == 2)
           os << es.corsi[i]->Nome() << ' ' << es.corsi[i]->CorsoDiStudi() << ' '
              << es.corsi[i]->SecondoAppello()->DataProva() << ' ' << es.corsi[i]->SecondoAppello()-
       }
```

```
   return os;
}

istream& operator>>(istream& is, SessioneEsami& es)
{
  string buffer, nome, cs, aula;
  unsigned num_corsi, num_corsi_mutuati, num_esami, i;

  Data data;
  int c;

  es.DeallocaVettori();

  is >> buffer >> es.quadrimestre;
  is >> buffer >> num_corsi;
  es.corsi.resize(num_corsi);
  for (i = 0; i < num_corsi; i++)
    {
      es.corsi[i] = new Corso();
      is >> *(es.corsi[i]);
    }
  is >> buffer >> num_corsi_mutuati;
  es.corsi_mutuati.resize(num_corsi_mutuati);
  for (i = 0; i < num_corsi_mutuati; i++)
    {
      es.corsi_mutuati[i] = new CorsoMutuato();
      is >> *(es.corsi_mutuati[i]) >> nome >> cs;
      c = es.CercaCorso(nome,cs);
      assert (c != -1);
      es.corsi_mutuati[i]->InserisciMutuante(es.corsi[c]);
    }
  is >> buffer >> num_esami;
  for (i = 0; i < num_esami; i++)
    {
      is >> nome >> cs >> data >> aula;
      c = es.CercaCorso(nome,cs);
      assert (c != -1);
      es.corsi[c]->InserisciAppello(new Esame(data, aula));
    }
  return is;
}
```

Soluzione esercizi 3 + driver

```
// File DriverEsami.cpp

#include <fstream>
#include "SessioneEsami.h"

bool VerificaConsistenza(const SessioneEsami& se);

int main()
{
  string nome_file("Settembre.txt");
  ifstream is(nome_file.c_str());
  SessioneEsami se(2);
  int scelta;
  string nome, cs, nome2, cs2, docente, aula;
  unsigned quad;
```

```
Data d;

is >> se;
is.close();

SessioneEsami se2 = se;

do
  {
    cout << "Menu : " << endl
         << "  (1) Inserisci corso" << endl
         << "  (2) Inserisci corso mutuato" << endl
         << "  (3) Inserisci esame" << endl
         << "  (4) Elimina corso" << endl
         << "  (5) Elimina corso mutuato" << endl
         << "  (6) Verifica consistenza" << endl
         << "  (7) Occupazione aule" << endl
         << "  (8) Stampa Stato Sessione" << endl
         << "  (0) Esci" << endl
         << " Scelta : ";
    cin >> scelta;
    switch (scelta)
      {
      case 1:
        {
          cout << "Nome corso : ";
          cin >> nome;
          cout << "Nome corso di studi : ";
          cin >> cs;
          cout << "Docente : ";
          cin >> docente;
          cout << "Quadrimestre: ";
          cin >> quad;
          se.InserisciCorso(nome,cs,docente,quad);
          break;
        }
      case 2:
        {
          cout << "Nome corso mutuato: ";
          cin >> nome;
          cout << "Nome corso di studi : ";
          cin >> cs;
          cout << "Nome corso mutuante: ";
          cin >> nome2;
          cout << "Nome corso di studi del corso mutuante : ";
          cin >> cs2;
          se.InserisciCorsoMutuato(nome,cs, nome2, cs2);
          break;
        }
      case 3:
        {
          cout << "Nome corso: ";
          cin >> nome;
          cout << "Nome corso di studi: ";
          cin >> cs;
          cout << "Data appello: ";
          cin >> d;
          cout << "Aula: ";
          cin >> aula;
```

```
            se.InserisciAppelloCorso(nome,cs,d,aula);
            break;
          }
        case 4:
          {
            cout << "Nome corso : ";
            cin >> nome;
            cout << "Nome corso di studi : ";
            cin >> cs;
            se.EliminaCorso(nome,cs);
            break;
          }
        case 5:
          {
            cout << "Nome corso mutuato: ";
            cin >> nome;
            cout << "Nome corso di studi : ";
            cin >> cs;
            se.EliminaCorsoMutuato(nome,cs);
            break;
          }
        case 6:
          {
            if (VerificaConsistenza(se))
              cout << "Sessione consistente!" << endl;
            else
              cout << "Sessione inconsistente!" << endl;
            break;
          }
        case 7:
          {
            cout << "Data: ";
            cin >> d;
            vector<string> aule = se.AuleOccupate(d);
            if (aule.size() == 0)
              cout << "Nessun'aula occupata";
            else
              for (unsigned i = 0; i < aule.size(); i++)
                cout << aule[i] << ' ';
            cout << endl;
            break;
          }
          case 8:
          {
            cout << se << endl;
            break;
          }
        }
    }
  while (scelta != 0);
  ofstream os(nome_file.c_str());
  os << se;
  os.close();
  cerr << se2;
}

bool VerificaConsistenza(const SessioneEsami& se)
{
  for (unsigned i = 0; i < se.Corsi(); i++)
```

```
    {
      if (se.VediCorso(i)->Quadrimestre() == se.Quadrimestre())
        {
          if (se.VediCorso(i)->NumAppelli() != 2)
            return false;
        }
      else
        {
          if (se.VediCorso(i)->NumAppelli() != 1)
            return false;
        }
    }
  return true;
}
```

Soluzione compito del 21 dicembre 2005

Soluzione esercizio 1

L'associazione Connessione viene rappresentata da un vettore di puntatori ad oggetti di classe `Locale`. La relazione è realizzata con responsabilità doppia. Essendo simmetrica, una rappresentazione con responsabilità singola sarebbe alquanto complessa e artificiosa.

La responsabilità sull'associazione Composizione viene invece attribuita alla classe `Appartamento`.

```
// File Locale.h
#ifndef LOCALE_H
#define LOCALE_H

#include <iostream>
#include <vector>

using namespace std;

class Locale
{
  friend ostream& operator<<(ostream& os, const Locale& l);
  friend istream& operator>>(istream& is, Locale& l);
  friend bool operator==(const Locale& l1, const Locale& l2);
 public:
  Locale(string c, string d, float mq);
  string Codice() const { return codice; }
  string Descrizione() const { return descrizione; }
  float MetriQuadri() const { return metri_quadri; }
  void SetDescrizione(string d) { descrizione = d; }
  void InserisciConnessione(Locale* l);
  void RimuoviConnessione(Locale* l);
  unsigned NumeroConnessioni() const { return connessioni.size(); }
  Locale* VediConnesso(unsigned i) const { return connessioni[i]; }
 private:
  string codice;
  string descrizione;
  float metri_quadri;
  vector<Locale*> connessioni;
  int CercaConnessione(Locale* l) const;
};

#endif
```

```
// File Appartamento.h
#ifndef APPARTAMENTO_H
#define APPARTAMENTO_H

#include <iostream>
#include <vector>
#include "Locale.h"

using namespace std;

class Appartamento
{
  friend ostream& operator<<(ostream& os, const Appartamento& l);
  friend istream& operator>>(istream& is, Appartamento& l);
 public:
  Appartamento(string ind, string n);
  string Indirizzo() const { return indirizzo; }
  string Nome() const { return nome; }
  float MetriQuadri() const;
  unsigned NumeroLocali() const { return locali.size(); }
  Locale* VediLocale(unsigned i) const { return locali[i]; }
  string StatoLavori() const;
  bool Appartiene(Locale* l) const { return CercaLocale(l) != -1; }
  void InserisciLocale(Locale* l);
  void RimuoviLocale(Locale* l);
  void RimuoviLocaleESueConnessioni(Locale* l);
  bool RendiPronto();
  void NuoviLavori();
  void Consegna();
 private:
  unsigned stato;
  string indirizzo;
  string nome;
  vector<Locale*> locali;
  int CercaLocale(Locale* l) const;
  bool Chiuso() const;
};

#endif
```

Soluzione esercizio 2

```
// File Locale.cpp

#include "Locale.h"
#include <cassert>

Locale::Locale(string c, string d, float mq)
  : codice(c), descrizione(d)
{
  metri_quadri = mq;
}

void Locale::InserisciConnessione(Locale* l)
{
  assert (l != this);
  int i = CercaConnessione(l);
  assert (i == -1);
  connessioni.push_back(l);
```

```
  l->connessioni.push_back(this);
}

void Locale::RimuoviConnessione(Locale* l)
{
  int i = CercaConnessione(l);
  if (i != -1)
    {
      int j = l->CercaConnessione(this);
      l->connessioni.erase(l->connessioni.begin() + j);
      connessioni.erase(connessioni.begin() + i);
    }
}

int Locale::CercaConnessione(Locale* l) const
{
  for (unsigned i = 0; i < connessioni.size(); i++)
    if (connessioni[i] == l)
      return i;
  return -1;
}

ostream& operator<<(ostream& os, const Locale& l)
{
  os << l.descrizione << ' ' << l.metri_quadri;
  return os;
}

istream& operator>>(istream& is, Locale& l)
{
  is >> l.descrizione >> l.metri_quadri;
  return is;
}

bool operator==(const Locale& l1, const Locale& l2)
{
  return l1.codice == l2.codice;
}
```

```
// File Appartamento.cpp
#include "Appartamento.h"
#include <cassert>

Appartamento::Appartamento(string ind, string n)
  : indirizzo(ind), nome(n)
{ stato = 0; }

void Appartamento::InserisciLocale(Locale* l)
{
  assert(stato == 0);
  assert(CercaLocale(l) == -1);
  locali.push_back(l);
}

void Appartamento::RimuoviLocale(Locale* l)
{
  int i = CercaLocale(l);
  assert(stato == 0);
  assert (i != -1);
  locali.erase(locali.begin() + i);
```

```
}

void Appartamento::RimuoviLocaleESueConnessioni(Locale* l)
{
  int i = CercaLocale(l);
  assert(stato == 0);
  assert (i != -1);
  for (unsigned j = 0; j < l->NumeroConnessioni(); j++)
    if (CercaLocale(l->VediConnesso(j)) != -1)
      l->VediConnesso(j)->RimuoviConnessione(l);
  locali.erase(locali.begin() + i);
}

float Appartamento::MetriQuadri() const
{
  float metri_quadri = 0.0;
  for (unsigned j = 0; j < locali.size(); j++)
    metri_quadri += locali[j]->MetriQuadri();
  return metri_quadri;
}

int Appartamento::CercaLocale(Locale* l) const
{
  for (unsigned j = 0; j < locali.size(); j++)
    if (*(locali[j]) == *l)
      return j;
  return -1;
}

string Appartamento::StatoLavori() const
{
  if (stato == 0)
    return "In preparazione";
  else if (stato == 1)
    return "Pronto";
  else if (stato == 2)
    return "Consegnato";
  else
    {
      assert(false);
      return "Errore";
    }
}

bool Appartamento::RendiPronto()
{
  assert(stato == 0);
  if (Chiuso())
    {
      stato = 1;
      return true;
    }
  else
    return false;
}

void Appartamento::NuoviLavori()
{
  assert(stato == 1);
```

```
  stato = 0;
}

void Appartamento::Consegna()
{
  assert(stato == 1);
  stato = 2;
}

bool Appartamento::Chiuso() const
{ // verifica che ciascun locale sia connesso solo con locali
  // dello stesso appartamento
  for (unsigned i = 0; i < locali.size(); i++)
      for (unsigned j = 0; j < locali[i]->NumeroConnessioni(); j++)
        if (!Appartiene(locali[i]->VediConnesso(j)))
          return false;
  return true;
}

ostream& operator<<(ostream& os, const Appartamento& ap)
{
  os << "Appartamento sito in " << ap.indirizzo << endl;
  os << "Denominato: " << ap.nome << endl;
  os << "Locali (" << ap.locali.size() << "): " << endl;
  for (unsigned i = 0; i < ap.locali.size(); i++)
    {
      os << ap.locali[i]->Codice() << ' ' << ap.locali[i]->Descrizione() << " (" << ap.locali[i]->Me
      for (unsigned j = 0; j < ap.locali[i]->NumeroConnessioni(); j++)
        os << "  connesso con " << ap.locali[i]->VediConnesso(j)->Codice() << endl;
    }
  return os;
}

istream& operator>>(istream& is, Appartamento& a)
{
  return is;
}
```

Soluzione esercizi 3 e 4 + driver

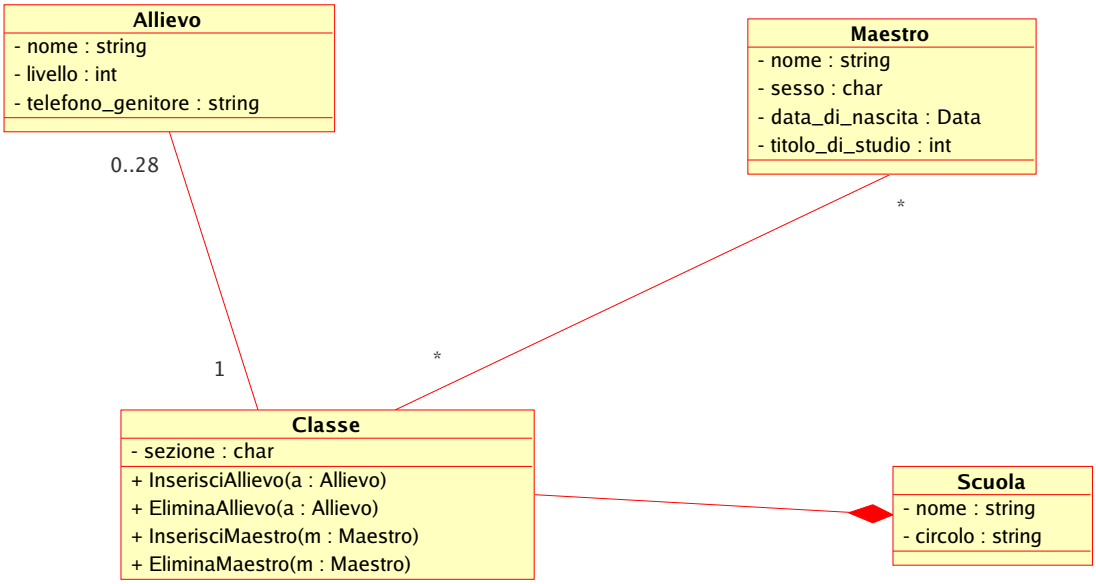
```
// File main.cpp

#include "Appartamento.h"

bool VerificaAbitabilita(const Appartamento& ap);

int main()
{
  Appartamento ap("Via Roma 22","Prestigiosa villa settecentesca");
  Locale l1("001", "bagno",6);
  Locale l2("002", "cucina",10);
  Locale l3("003", "salotto",18);
  Locale l4("004", "cameretta",11);
  Locale l5("005", "camera",11);

  l1.InserisciConnessione(&l5);
  l1.InserisciConnessione(&l4);
 //  l1.InserisciConnessione(&l1); fa fallire l'assert
```

```
  ap.InserisciLocale(&l1);
  ap.InserisciLocale(&l5);
  //  ap.RendiPronto();
  ap.InserisciLocale(&l4);

  cout << ap;
  if (VerificaAbitabilita(ap))
    cout << "Abitabile" << endl;
  else
    cout << "Non abitabile" << endl;

  ap.InserisciLocale(&l2);

  cout << ap;

  if (VerificaAbitabilita(ap))
    cout << "Abitabile" << endl;
  else
    cout << "Non abitabile" << endl;

  ap.RimuoviLocale(&l1);

  cout << ap << endl;
  if (ap.RendiPronto())
    cout << "Reso Pronto" << endl;
  else
    cout << "Impossibile, non chiuso" << endl;

  cout << ap.StatoLavori() << endl;
}

bool VerificaAbitabilita(const Appartamento& ap)
{ // verifica che sia presente esattamente una cucina e almeno un bagno
  unsigned bagni = 0, cucine = 0;
  for (unsigned i = 0; i < ap.NumeroLocali(); i++)
    {
      if (ap.VediLocale(i)->Descrizione() == "bagno")
        bagni++;
      else if (ap.VediLocale(i)->Descrizione() == "cucina")
        cucine++;
    }
  return cucine == 1 && bagni >= 1;
}
```

Soluzione compito del 11 dicembre 2006

Soluzione esercizio 1

Una classe può appartenere ad una sola scuola, è parte integrante di essa e non ha significato all'esterno di essa. Di conseguenza la relazione tra Scuola e Classe si configura come una composizione (part-of).



Soluzione esercizio 2

La relazione tra Scuola e Classe essendo di tipo part-of può essere realizzata con un vettore di oggetti (e non di puntatori ad oggetti). Questo semplifica il codice evitando di dover ricorrere ad allocazione dinamica di oggetti.

```
#ifndef ALLIEVO_H
#define ALLIEVO_H
#include <fstream>
#include <string>

using namespace std;

class Allievo
{
  friend bool operator==(const Allievo& a1, const Allievo& a2);
  friend ostream& operator<<(ostream& os, const Allievo& a);
 public:
  Allievo(string n, int l, string tg)
    : nome(n), telefono_genitore(tg)
  { livello = l; }
  string Nome() const { return nome; }
  unsigned Livello() const { return livello; }
  string NomeLivello() const;
  string TelefonoGenitore() const { return telefono_genitore; }
private:
  string nome;
  unsigned livello;
  string telefono_genitore;
};

#endif //ALLIEVO_H


#ifndef MAESTRO_H
#define MAESTRO_H
#include <string>
#include <iostream>
#include "../Utilities/Data.h"

using namespace std;
```

```
class Maestro
{
  friend bool operator==(const Maestro& m1, const Maestro& m2);
  friend ostream& operator<<(ostream& os, const Maestro& m);
public:
  Maestro(string n, char s, Data dn, string ts)
    : nome(n), data_di_nascita(dn),
      titolo_di_studio(ts) { sesso = s; }
  string Nome() const { return nome; }
  char Sesso() const { return sesso; }
  Data DataDiNascita() const { return data_di_nascita; }
  string TitoloDiStudio() const { return titolo_di_studio; }
private:
   string nome;
   char sesso;
   Data data_di_nascita;
   string titolo_di_studio;
};

#endif //MAESTRO_H


#ifndef CLASSE_H
#define CLASSE_H
#include <string>
#include <vector>
#include "allievo.h"
#include "maestro.h"

using namespace std;

class Classe
{
  friend ostream& operator<<(ostream& os, const Classe& c);
 public:
  Classe(char s) : MAX_ALLIEVI(28) { sezione = s; }
  char Sezione() const { return sezione; }
  Allievo* VediAllievo(unsigned i) const { return allievi[i]; }
  Maestro* VediMaestro(unsigned i) const { return maestri[i]; }
  unsigned Allievi() const { return allievi.size(); }
  unsigned Maestri() const { return maestri.size(); }

  bool Piena() const { return allievi.size() == MAX_ALLIEVI; }

  unsigned IndiceAllievoTrasferibile() const;

  void InserisciAllievo (Allievo* a);
  void EliminaAllievo (Allievo* a);
  void InserisciMaestro (Maestro* m);
  void EliminaMaestro (Maestro* m);

   int CercaAllievo(Allievo* a);
   int CercaMaestro(Maestro* m);

private:
   char sezione;
   vector<Allievo*> allievi;
   vector<Maestro*> maestri;
   unsigned MAX_ALLIEVI;
```

```
};
#endif //CLASSE_H


#ifndef SCUOLA_H
#define SCUOLA_H
#include <string>
#include <vector>
#include "classe.h"

class Scuola
{
  friend ostream& operator<<(ostream& os, const Scuola& sc);
public:
  Scuola(string n, string c) : nome(n), circolo(c) {}
  string Nome() const { return nome; }
  string Circolo() const { return circolo; }
  unsigned Classi() const { return classi.size(); }
  Classe VediClasse(unsigned i) const { return classi[i]; }

  void InserisciAllievo(char c, Allievo* a);
  void EliminaAllievo(Allievo* a);
  void AssegnaMaestro(char c, Maestro* m);
  void RevocaMaestro(Maestro* m);
  void CreaClasse();
  unsigned TotaleAllievi();

private:
  string nome;
  string circolo;
  vector <Classe> classi;
  unsigned ClassePiuNumerosa();
};
#endif //SCUOLA_H
```

Soluzione esercizio 3

```
#include "allievo.h"

string Allievo::NomeLivello() const {
    if (livello == 0) return "piccolo";
    else if (livello == 1) return "medio";
    else if (livello == 2) return "grande";
    else return "livello sconosciuto";
 }

bool operator==(const Allievo& a1, const Allievo& a2)
{ return a1.nome == a2.nome; }

ostream& operator<<(ostream& os, const Allievo& a)
{
  os << a.nome << ' ' << a.livello << ' ' << a.telefono_genitore;
  return os;
}

#include "maestro.h"

bool operator==(const Maestro& m1, const Maestro& m2)
{ return m1.nome == m2.nome; }
```

```
ostream& operator<<(ostream& os, const Maestro& m)
{
  os << m.nome << ' ' << m.sesso << ' ' << m.data_di_nascita << ' '
     << m.titolo_di_studio;
  return os;
}
```

```
#include <cassert>
#include "classe.h"

void Classe::InserisciAllievo (Allievo* a)
{
  assert (allievi.size() < MAX_ALLIEVI);
  assert (CercaAllievo(a) == -1);
  allievi.push_back(a);
}

void Classe::EliminaAllievo(Allievo* a)
{
  int i = CercaAllievo(a);
  assert (i != -1);
  allievi.erase(allievi.begin() + i);
}

void Classe::InserisciMaestro (Maestro* m)
{
  assert (CercaMaestro(m) == -1);
  maestri.push_back(m);
}

void Classe::EliminaMaestro (Maestro* m)
{
  int i = CercaMaestro(m);
  assert (i != -1);
  maestri.erase(maestri.begin() + i);

}

unsigned Classe::IndiceAllievoTrasferibile() const
{
  const unsigned NUM_LIVELLI = 3;
  unsigned i, l, livello_max = 0;
  vector<unsigned> num_allievi(NUM_LIVELLI, 0);

  assert(allievi.size() > 0);

  // crea il vettore delle frequenze
  for (i = 0; i < allievi.size(); i++)
    num_allievi[allievi[i]->Livello()]++;

  // cerca il massimo del vettore delle frequenze
  for (l = 1; l < NUM_LIVELLI; l++)
    if (num_allievi[l] > num_allievi[livello_max])
      livello_max = l;

  // cerca il primo allievo del livello massimo
  for (i = 0; i < allievi.size(); i++)
    if (allievi[i]->Livello() == livello_max)
      return i;
```

```
  // non si dovrebbe arrivare qui
  assert(false);
  return 0;
}

int Classe::CercaAllievo(Allievo* a)
{
  for (unsigned i = 0; i < allievi.size(); i++)
    if (*a == *(allievi[i]))
      return i;
  return -1;
}

int Classe::CercaMaestro(Maestro* m)
{
  for (unsigned i = 0; i < maestri.size(); i++)
    if (*m == *(maestri[i]))
      return i;
  return -1;
}

ostream& operator<<(ostream& os, const Classe& c)
{
  unsigned i;
  os << "Classe " << c.sezione << endl;
  os << "Allievi : " << endl;
  for (i = 0; i < c.allievi.size(); i++)
    os << *(c.allievi[i]) << endl;
  os << "Maestri : " << endl;
  for (i = 0; i < c.maestri.size(); i++)
    os << *(c.maestri[i]) << endl;
  return os;
}


#include "scuola.h"

void Scuola::InserisciAllievo(char c, Allievo* a)
{
  unsigned i = c - 'A'; // indice della classe
  unsigned j;

  assert (i >= 0 && i < classi.size());
  assert (!classi[i].Piena());

  for (j = 0; j < classi.size(); j++)
    assert (classi[j].CercaAllievo(a) == -1);

  classi[i].InserisciAllievo(a);
}

void Scuola::EliminaAllievo(Allievo* a)
{
  unsigned j;
  int i;

  for (j = 0; j < classi.size(); j++)
    {
      i = classi[j].CercaAllievo(a);
```

```
      if (i != -1)
        {
          classi[j].EliminaAllievo(a);
          return;
        }
    }

  assert(false);
}

void Scuola::AssegnaMaestro(char c, Maestro* m)
{
  int i = c - 'A'; // indice della classe
  assert (i >= 0 && i < int(classi.size()));
  classi[i].InserisciMaestro(m);
}

void Scuola::RevocaMaestro(Maestro* m)
{
  for (unsigned i = 0; i < classi.size(); i++)
    if (classi[i].CercaMaestro(m) != -1)
      classi[i].EliminaMaestro(m);
}

void Scuola::CreaClasse()
{
  char c = classi.size() + 'A';
  Classe cl(c);
  Allievo* a;
  unsigned i, j;

  if (TotaleAllievi() > 1)
    {
      i = ClassePiuNumerosa();
      j = classi[i].IndiceAllievoTrasferibile();
      a = classi[i].VediAllievo(j);
      classi[i].EliminaAllievo(a);
      cl.InserisciAllievo(a);
    }
  classi.push_back(cl);
}

unsigned Scuola::ClassePiuNumerosa()
{
  unsigned max = 0;
  for (unsigned i = 1; i < classi.size(); i++)
    if (classi[i].Allievi() > classi[max].Allievi())
      max = i;
  return max;
}

unsigned Scuola::TotaleAllievi()
{
  unsigned tot = 0;
  for (unsigned i = 0; i < classi.size(); i++)
    tot += classi[i].Allievi();
  return tot;
}
```

```
ostream& operator<<(ostream& os, const Scuola& sc)
{
  os << "Stato Scuola : " << endl;
  for (unsigned i = 0; i < sc.classi.size(); i++)
    os << sc.classi[i] << endl;
  return os;
}
```

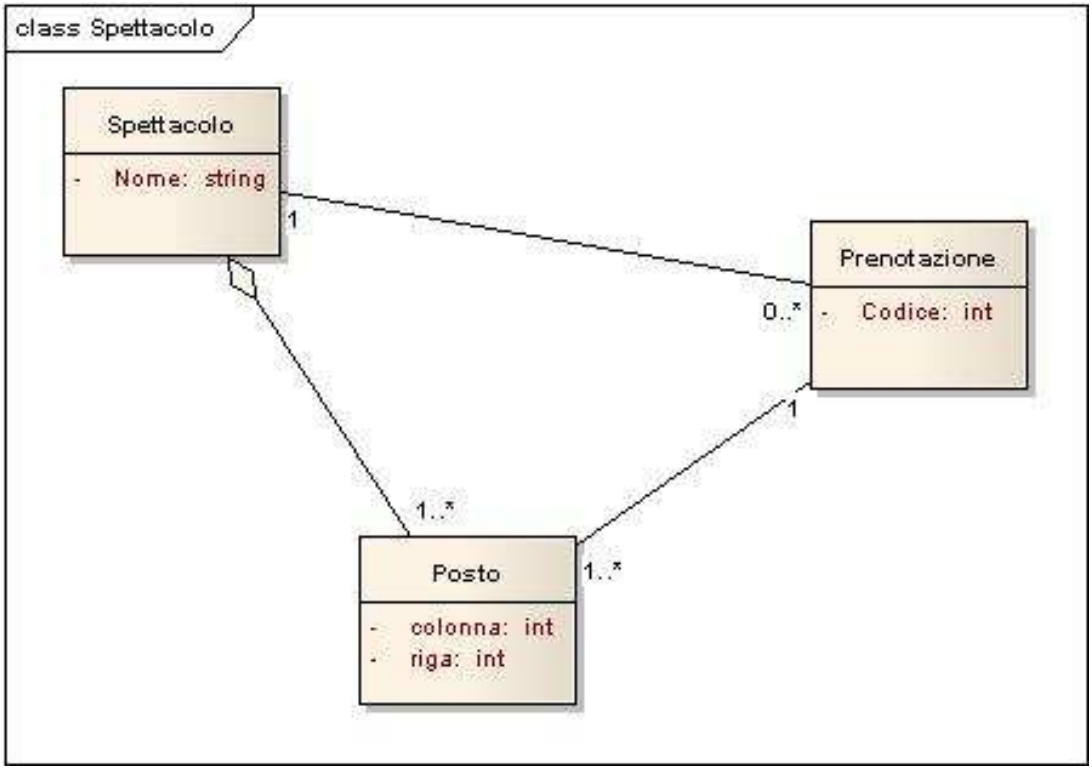
Soluzione esercizi 4 + driver

```
#include "scuola.h"

vector<bool> ClassiOmogenee(const Scuola& s);

int main()
{
  Allievo a1("Mario Rossi", 0, "555 3459");
  Allievo a2("Mario Gialli", 1, "555 3460");
  Allievo a3("Maria Rossi", 2, "555 3459");
  Allievo a4("Maria Gialli", 1, "555 3460");
  Allievo a5("Marta Gialli", 1, "555 3460");
  Allievo a6("Francesca Gialli", 1, "555 3460");
  Allievo a7("Francesco Gialli", 2, "555 3460");
  Allievo a8("Francescu Gialli", 2, "555 3460");

  Maestro m1("Francesco Bianchi", 'm', Data(13,12,1971), "Laurea ...");
  Maestro m2("Francesca Bianchi", 'f', Data(14,12,1971), "Laurea ...");

  Scuola sc("G. Carducci", "IV, Udine");
  sc.CreaClasse();
  sc.InserisciAllievo('A', &a1);
  sc.InserisciAllievo('A', &a2);
  sc.CreaClasse();
  cerr << sc << endl;

  sc.InserisciAllievo('B', &a3);
  sc.CreaClasse();
  cerr << sc << endl;

  sc.InserisciAllievo('C', &a4);
  sc.InserisciAllievo('C', &a5);
  sc.InserisciAllievo('C', &a6);
  sc.CreaClasse();

  sc.AssegnaMaestro('A', &m1);
  sc.AssegnaMaestro('C', &m1);
  sc.AssegnaMaestro('A', &m2);

  sc.InserisciAllievo('B', &a7);
  sc.InserisciAllievo('D', &a8);

  cerr << sc << endl;

  vector<bool> v = ClassiOmogenee(sc);

  for (unsigned i = 0; i < v.size(); i++)
    cerr << sc.VediClasse(i).Sezione() << " : " << (v[i]?"Si":"No") << endl;
```

```
  return 0;
}

vector<bool> ClassiOmogenee(const Scuola& s)
{
  vector<bool> classi_omogenee(s.Classi(), true);

  for (unsigned i = 0; i < s.Classi(); i++)
    {
      Classe c = s.VediClasse(i);
      if (c.Allievi() > 0)
        {
          unsigned l = c.VediAllievo(0)->Livello();
          for (unsigned j = 1; j < c.Allievi(); j++)
            if (c.VediAllievo(j)->Livello() != l)
              {
                classi_omogenee[i] = false;
                break;
              }
        }
    }
  return classi_omogenee;
}
```

Soluzione compito del 10 dicembre 2007

Soluzione esercizio 1



Soluzione esercizio 2

```
#ifndef CINEMA_H
#define CINEMA_H
```

```
#include <vector>
#include <string>
#include <iostream>

using namespace std;

class Posto
{
  friend ostream& operator<<(ostream& os, const Posto& p);
public:
  Posto(unsigned f, unsigned c) { fila = f; colonna = c; }
  unsigned Fila() const { return fila; }
  unsigned Colonna() const { return colonna; }
private:
  unsigned fila, colonna;
};

class Prenotazione
{
  friend ostream& operator<<(ostream& os, const Prenotazione& p);
public:
  Prenotazione(unsigned c);
  unsigned Codice() const { return codice; }
  unsigned Posti() const { return posti.size(); }
  void AggiungiPosto(const Posto& p);
  Posto VediPosto(unsigned i) const { return posti[i]; }
  vector<Posto> VediPosti() const { return posti; }
private:
  unsigned codice;
  vector<Posto> posti;
};

class Spettacolo
{
  friend ostream& operator<<(ostream& os, const Spettacolo& s);
public:
  Spettacolo(string n, unsigned f, unsigned c);
  string Nome() const { return nome; }
  int AggiungiPrenotazione(unsigned num_posti);
  int AggiungiPosto(const Posto& posto);
  bool RimuoviPrenotazione(unsigned cod);
  vector<Posto> VediPrenotazione(unsigned cod);
private:
  int CercaPrenotazione(unsigned cod);
  string nome;
  unsigned file, colonne, num_posti_liberi, nuovo_codice;
  vector<Prenotazione> prenotazioni;
  vector<vector<bool> > posti_liberi;
};
#endif
```

Soluzione esercizio 3

```
#include "spettacolo.h"
#include <cassert>

Prenotazione::Prenotazione(unsigned c)
{
```

```
  codice = c;
}

void Prenotazione::AggiungiPosto(const Posto& p)
{
  posti.push_back(p);
}

Spettacolo::Spettacolo(string n, unsigned f, unsigned c)
  : nome(n), posti_liberi(f,vector<bool>(c,true))
{
  file = f;
  colonne = c;
  num_posti_liberi = file * colonne;
  nuovo_codice = 0;
}

int Spettacolo::AggiungiPrenotazione(unsigned num_posti)
{
  unsigned i = 0, j = 0, posti = 0;

  if (num_posti > num_posti_liberi)
    return -1;
  else
    {
      nuovo_codice++;
      Prenotazione pren(nuovo_codice);
      while (posti < num_posti)
        {
          if (posti_liberi[i][j])
            {
              pren.AggiungiPosto(Posto(i,j));
              posti_liberi[i][j] = false;
              num_posti_liberi--;
              posti++;
            }
          j++;
          if (j == colonne)
            {
              i++;
              j = 0;
            }
        }
      prenotazioni.push_back(pren);
      return nuovo_codice;
    }
}


int Spettacolo::AggiungiPosto(const Posto& posto)
{
  if (posti_liberi[posto.Fila()][posto.Colonna()])
    {
      nuovo_codice++;
      Prenotazione pren(nuovo_codice);
      pren.AggiungiPosto(posto);
      posti_liberi[posto.Fila()][posto.Colonna()] = false;
      prenotazioni.push_back(pren);
      return nuovo_codice;
```

```
    }
  else
    return -1;
}

bool Spettacolo::RimuoviPrenotazione(unsigned cod)
{
  unsigned j;
  int i = CercaPrenotazione(cod);

  if (i != -1)
    {
      for (j = 0; j < prenotazioni[i].Posti(); j++)
        {
          Posto p = prenotazioni[i].VediPosto(j);
          posti_liberi[p.Fila()][p.Colonna()] = true;
        }
      prenotazioni.erase(prenotazioni.begin() + i);
      return true;
    }
  else
    return false;
}

vector<Posto> Spettacolo::VediPrenotazione(unsigned cod)
{
  int i = CercaPrenotazione(cod);
  if (i == -1)
    return vector<Posto>();
  else
    return prenotazioni[i].VediPosti();
}

int Spettacolo::CercaPrenotazione(unsigned cod)
{
  unsigned i;
  for (i = 0; i < prenotazioni.size(); i++)
    if (prenotazioni[i].Codice() == cod)
      return i;
  return -1;
}

ostream& operator<<(ostream& os, const Posto& p)
{
  os << '(' << p.fila << ',' << p.colonna << ')';
  return os;
}

ostream& operator<<(ostream& os, const Prenotazione& p)
{
  unsigned i;
  os << p.codice << " : ";
  for (i = 0; i < p.posti.size(); i++)
    os << p.posti[i] << " ";
  return os;
}

ostream& operator<<(ostream& os, const Spettacolo& s)
{
```

```
  unsigned i,j;
  os << "Sistema di prenotazione per lo spettacolo: " << s.nome << endl;
  os << endl << "Mappa dei posti in sala: " << endl;
  for (i = 0; i < s.file; i++)
    {
      for (j = 0; j < s.colonne; j++)
        os << (s.posti_liberi[i][j] ? '-' : 'X');
      os << endl;
    }

  os << endl << "Elenco prenotazioni: " << endl;
  for (i = 0; i < s.prenotazioni.size(); i++)
    os << "Prenotazione " << s.prenotazioni[i] << endl;
  return os;
}
```