ALGORITMI E CALCOLATORI A.A. 2020/2021

**GESTIONE UNIVERSITÀ E PROGRAMMAZIONE ESAMI**

Il seguente progetto si pone l'obiettivo di occuparsi della gestione delle risorse e dei dati inerenti l'università con annessa allocazione delle giornate di esame. Infatti la problematica principale nella gestione di una università è proprio quella che riguarda la gestione dell’allocazione delle giornate di esame. Tale gestione infatti richiede che, a fronte delle regole di ateneo, si organizzino le date degli esami in modo da non violare alcun regolamento interno all'università stessa.

Tale progetto si basa infatti sulla creazione e definizione di una base dati iniziale che, mediante dei comandi da terminale, può essere modificata mediante l’inserimento di nuovi dati o l’aggiornamento di quelli già esistenti.

Spiegazione del fatto che si hanno dei file di input come database, delle mappe in memoria e dei file di output.

Spiegazione TOP-DPWN delle classi: università - corsi di studio – corsi – specific year – studenti – professori

citando l’EREDITARIETA’

Spiegazione Parse

Spiegazione eccezioni

I metodi della classe “University” vengono riportati di seguito:

* **readStudents**

prototipo: void University::readStudents()

Questo metodo si occupa della lettura del database degli studenti denominato: db\_studenti.txt .

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “InteroStudente” in modo che ogni elemento del vector contenga l’informazione presente nella riga compresa tra due caratteri separatori “ ; ”. Successivamente mediante un oggetto stringstream (ovvero un oggetto stream che consente di effettuare delle operazioni sulle stringhe) viene salvata solo la matricola dello studente, precedentemente contenuta insieme al carattere “s” nel primo elemento di “InteroStudente”.

Infine si controlla se la matricola associata allo studente corrente è già esistente in memoria; in tal caso viene lanciata una eccezione di avviso, diversamente si procede con il salvataggio nella mappa in memoria dello studente e delle relative informazioni contenute nei diversi campi del vector “InteroStudente”:

* “InteroStudente[0]” contiene s-matricola;
* “InteroStudente[1]” contiene uno o più nomi dello studente (in quanto la funzione splittedLine è stata applicata rispetto al carattere “ ;”;
* “InteroStudente[2]” contiene uno o più cognomi dello studente (in quanto la funzione splittedLine è stata applicata rispetto al carattere “ ;”;
* “InteroStudente[3]” contiene l’indirizzo e-mail dello studente;

Vista la presenza della “modalità versioning” si lascia anche la possibilità di caricare le diverse informazioni aggiuntive riguardanti gli studenti; siccome tale modalità si verifica nel momento in cui \_version = 2 si distingue mediante un if il caricamento delle info base dal caricamento dettato dalla modalità versioning. Pertanto alle info precedenti sugli studenti si aggiungono le tre seguenti:

* “InteroStudente [4]” contiene la data di nascita;
* “InteroStudente [5]” contiene l’anno di immatricolazione;
* “InteroStudente [6]” contiene l’indirizzo di casa;

La chiave della mappa degli studenti è quindi la matricola dello studente (privata del carattere “s”) in quanto la matricola è univoca per ogni studente.

Il metodo si conclude con la chiusura del file database degli studenti “db\_studenti.txt”.

* **readVersion**

prototipo: void University::readVersion()

Questo metodo si occupa della lettura del file “versione.txt” contenente il numero corrispondente alla versione impostata per la “modalità versioning”.

Si ricorda che si scrive su un apposito file, “versione.txt”, la versione adottata affinché qualora il programma dovesse terminare si tiene traccia del contenuto che possedeva la variabile “\_version”; pertanto risulta fondamentale andare a leggere la versione che il programma stava utilizzando prima dello spegnimento, da qui l’importanza del metodo readVersion().

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso a causa della sua inesistenza, si procede con la lettura del numero corrispondente alla modalità versioning contenuta nell’unica linea presente nel file; il risultato viene salvato nella stringa “line” e poi convertito in un intero e destinato alla variabile “\_version”.

Come di buona norma il metodo si conclude con la chiusura del file mediante l’apposito metodo close.

* **readProfessor**

prototipo: void University::readProfessor()

Questo metodo si occupa della lettura del database dei professori denominato: db\_professori.txt .

Tale metodo si sviluppa in modo analogo al precedente degli studenti.

Si riportano di seguito le informazioni contenute nei diversi campi del vector “InteroProfessore”:

* “InteroProfessore [0]” contiene d-matricola (anche in questo caso il carattere “d”;
* “InteroProfessore [1]” contiene uno o più nomi del professore (in quanto la funzione splittedLine è stata applicata rispetto al carattere “ ;”;
* “InteroProfessore [2]” contiene uno o più cognomi del professore (in quanto la funzione splittedLine è stata applicata rispetto al carattere “ ;”;
* “InteroProfessore [3]” contiene l’indirizzo e-mail del professore;

Vista la presenza della “modalità versioning” si lascia anche la possibilità di caricare le diverse informazioni aggiuntive riguardanti i professori; siccome tale modalità si verifica nel momento in cui \_version = 2 si distingue mediante un if il caricamento delle info base dal caricamento dettato dalla modalità versioning. Pertanto alle info precedenti sugli studenti si aggiungono le tre seguenti:

* “InteroProfessore [4]” contiene la data di nascita;
* “InteroProfessore [5]” contiene l’anno di immissione di ruolo;
* “InteroProfessore [6]” contiene l’indirizzo di casa;

La chiave della mappa dei professori è quindi la matricola del professore (privata del carattere “d”) in quanto la matricola è univoca per ogni professore.

Il metodo si conclude con la chiusura del file database dei professori “db\_ professori.txt”.

* **readClassroom**

prototipo: void University::readClassroom()

Questo metodo si occupa della lettura del database delle aule denominato: db\_aule.txt .

Tale metodo si sviluppa in modo analogo al precedente degli studenti.

Si riportano di seguito le informazioni contenute nei diversi campi del vector “InteraClasse”:

* “InteraClasse [0]” contiene A-codice identifico dell’aula;
* “InteraClasse [1]” contiene la stringa che specifica il tipo di aula: “A” in caso di aula, “L” in caso di laboratorio;
* “InteraClasse [2]” contiene il nome dell’aula;
* “InteraClasse [3]” contiene la capienza massima dell’aula;
* “InteraClasse [4]” contiene la capienza massima dell’aula se adibita per un esame;

La chiave della mappa delle aule è quindi la matricola delle aule (privata del carattere “A”) in quanto la matricola è univoca per ogni aula.

Il metodo si conclude con la chiusura del file database dei professori “db\_ aule.txt”.

* **readStudyCourse**

prototipo: void University::readStudyCourse()

Questo metodo si occupa della lettura del database dei corsi di studio denominato: db\_corsi\_studio.txt .

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “InteroCorsoDiStudi” in modo che ogni elemento del vector contenga l’informazione presente nella riga compresa tra due caratteri separatori “ ; ”. Successivamente mediante un oggetto stringstream viene salvato solo il codice identificativo del corso di studio, precedentemente contenuto insieme al carattere “C” nel primo elemento di “InteroCorsoDiStudi”.

In particolare vengono riportate di seguito le informazioni contenute nel vector “InteroCorsoDiStudi”:

* “InteroCorsoDiStudi [0]” contiene C-codice identificativo del corso di studio;
* “InteroCorsoDiStudi [1]” contiene le due stringhe che specificano il tipo di corso di studio: BS in caso di laurea triennale, MS in caso di laurea magistrale;
* “InteroCorsoDiStudi [2]” contiene l’elenco dei codici identificativi dei diversi corsi raggruppati per semestre all’interno delle parentesi graffe;
* “InteroCorsoDiStudi [3]” contiene l’elenco dei codici identificativi dei corsi spenti separati da una virgola;

Visto il contenuto dell’elemento “InteroCorsoDiStudi [2]” nasce l’esigenza di separare i diversi corsi per ogni semestre. Si usa la funzione posCurlyBrackets la quale, passandole “InteroCorsoDiStudi [2]”, restituisce un vector di interi denominato “posSem” che contiene le diverse posizioni che occupano le parentesi graffe (aperte e chiuse) all’interno di “InteroCorsoDiStudi [2]”, posizione intesa come numero di caratteri che, a partire dal primo presente in “InteroCorsoDiStudi [2]”, si trova la parentesi graffa aperta o chiusa.

A questo punto, grazie ad un ciclo for, per ogni parentesi graffa aperta, che vista la formattazione del file “db\_corsi\_studio.txt“ si trovano nelle posizioni pari del vector “posSem” (motivo per cui l’indice del for viene incrementato di due ad ogni ciclo), si eseguono le seguenti operazioni :

1. si prende la posizione della parentesi graffa aperta, la si salva in una variabile di appoggio “posStart”;
2. si calcola la distanza “len” tra la posizione della “{“ e la posizione della successiva “}”; distanza intesa sempre in termini di caratteri presenti tra le due parentesi;
3. tramite il metodo delle stringhe “substr” si copiano in una stringa tutti i caratteri presenti nella stringa “InteroCorsoDiStudi [2]” che vanno dalla posizione presente in “posStart” (ovvero quella della parentesi graffa aperta) per un numero di caratteri pari al valore contenuto nella variabile “len”; il metodo “substr” restituisce così una stringa contenente i codici di tutti i corsi di un solo semestre; tale stringa viene poi copiata in coda, per ogni ciclo for, nel vector “semestri”.

Alla fine di questo for il vector “semestri” conterrà i corsi di ogni semestre disposti al suo interno in modo che ogni campo di “semestri” contiene tutti i corsi di un certo semestre nel seguente modo:

* “semestri [0]” contiene tutti i corsi del 1o anno 1o semestre;
* “semestri [1]” contiene tutti i corsi del 1o anno 2o semestre;
* “semestri [2]” contiene tutti i corsi del 2o anno 1o semestre;
* “semestri [3]” contiene tutti i corsi del 2o anno 2o semestre;
* “semestri [4]” contiene tutti i corsi del 3o anno 1o semestre;
* “semestri [5]” contiene tutti i corsi del 3o anno 2o semestre;

Si fa notare che il vector ”semestri” non è detto a priori che sia composto da sei campi in quanto questo dipende dal tipo di percorso di studio (triennale o magistrale) e anche dal numero di corsi presenti nel file “db\_corsi\_studio.txt”; quello appena riportato pertanto era solo un esempio generale con fine esplicativo.

Nel caso in cui la linea letta da file, “line”, contenga anche i corsi spenti, ovvero nel caso in cui il vector “InteroCorsoDiStudi” sia composto da un numero di campi pari a quattro (motivo per cui è presente un if con controllo sulla dimensione del vector pari a 4) allora si copiano in una stringa di appoggio mediante il metodo delle stringhe ”substr” tutti i caratteri presenti nel quarto campo di “InteroCorsoDiStudi” che vanno dal secondo carattere (in quanto il primo, vista la formattazione del file, è una parentesi quadra aperta) e fino al penultimo (in quanto l’ultimo, vista la formattazione del file, è una parentesi quadra chiusa); a questo punto l’elenco dei corsi spenti privato delle parentesi quadre (aperta e chiusa) viene passato alla funzione “splittedLine” insieme al carattere separatore, che in questo caso è la virgola “,”, la quale pertanto restituisce pertanto tutti i corsi spenti posizionati ognuno in un campo diverso del vector di stringhe “corsiSpenti”.

Dopo aver letto dalla riga tutte le informazioni inerenti un determinato corso di studio e sistemate in delle apposite variabili si passa alla creazione di un oggetto di appoggio denominato “SCourse” passando al relativo costruttore il codice identificativo del corso di studio e l’informazione inerente il tipo di corso di studio, ovvero percorso triennale o magistrale.

Con l’ausilio del metodo “addSemesterCourses”, appositamente creato per gli oggetti della classe “StudyCourse” (come dimostra il file “StudyCourse.h”), si riempie la mappa privata interna dei semestri dell’oggetto di appoggio poc’anzi creato, “SCourse”, la quale contiene i diversi corsi per ogni di semestre di ogni anno; pertanto mediante un ciclo for, per ogni stringa del vector “semestri” si procede come spiegato di seguito:

1. Si calcola l’anno associato al corso/i presente nella i-esima posizione in “semestri”; siccome il vector “semestri” contiene i corsi di ogni semestre disposti al suo interno in ordine crescente per semestre ovvero partendo dal 1o semestre del 1o anno fino al 2o semestre del 3o (o fino al 2o semestre del 2o anno in caso di laurea magistrale) è facile dedurre come l’anno sia pari al numero del semestre (incrementato di 1 perché la numerazione degli indici in un vector parte da 0) diviso 2 (in quanto ogni anno contiene due semestri);
2. Analogamente si calcola il semestre (se 1o o 2o) associato all’i-esimo corso/i che pertanto risulta essere uguale al resto dell’operazione che ha portato al calcolo dell’anno;
3. Si riempie la mappa privata dell’oggetto “SCourse” grazie al metodo “addSemesterCourses” al quale viene passato: l’anno associato all’i-esimo corso/i, il semestre (1o o 2o) associato all’i-esimo corso/i ed infine l’i-esimo corso/i;

Successivamente in caso di presenza di corsi spenti attraverso il metodo “addOffCourses”, appositamente creato per gli oggetti della classe “StudyCourse” (come dimostra il file “StudyCourse.h”), si riempie la lista di stringhe privata interna dei semestri spenti dell’oggetto di appoggio “SCourse”.

Come ultima operazione si riempie la mappa interna privata denominata “\_studyCourse” della classe “University” (rispettando la convenzione dell’anteporre un underscore al nome dei dati privati di una classe) con il metodo “insert” al quale viene passato come chiave della mappa il codice del corso e come value corrispondente l’oggetto di appoggio “SCourse”.

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

Tale metodo si conclude con la chiusura del file database dei corsi di studio “db\_corsi\_studio.txt”.

* **readCourse**

prototipo: void University::readCourse()

Questo metodo si occupa della lettura del database dei corsi denominato: db\_corsi.txt .

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “interoCorso” in modo che ogni elemento del vector contenga l’informazione presente nella riga compresa tra due caratteri separatori “ ; ”.

Il file “db\_corsi.txt” presenta una struttura secondo la quale:

* ogni riga che comincia con il carattere “c” contiene le info generali di quel corso;
* ogni riga che comincia con il carattere “a” contiene le info di quel corso specifiche per un certo anno accademico;

Pertanto si controlla il primo elemento del vector “interoCorso” (ovvero “interoCorso [0]”) e nel caso in cui fosse uguale alla stringa “c” si passa all’elaborazione delle info generali in particolare a seguire vengono riportate le info presenti nei diversi campi del vector “interoCorso”:

* “interoCorso [1]” contiene il codice identificativo del corso salvato poi in una variabile di appoggio denominata “lastReadCourse”;
* “interoCorso [2]” contiene il titolo del corso;
* “interoCorso [3]” contiene il numero di cfu del corso;
* “interoCorso [4]” contiene il numero di ore di lezione del corso;
* “interoCorso [5]” contiene il numero di ore di esercitazione del corso;
* “interoCorso [6]” contiene il numero di ore di laboratorio del corso;

Si riempie in seguito la mappa interna privata denominata “\_courses” della classe “University” (rispettando la convenzione dell’anteporre un underscore al nome dei dati privati di una classe) con il metodo “insert” al quale viene passato come chiave della mappa il codice del corso (contenuto nella variabile “lastReadCourse” e come value corrispondente l’oggetto della classe “Course”, inizializzato mediante costruttore al quale vengono passati (come da definizione presente nel file “Course.h”) i seguenti dati:

* codice identificativo del corso contenuto in “interoCorso [1]”;
* titolo del corso contenuto in “interoCorso [2]”;
* numero di cfu del corso contenuto in “interoCorso [3]” (convertito da stringa ad intero mediante il metodo delle stringhe stoi);
* numero delle ore di lezione del corso contenuto in “interoCorso [4]” (convertito da stringa ad intero mediante il metodo delle stringhe stoi);
* numero delle ore di esercitazione del corso contenuto in “interoCorso [5]” (convertito da stringa ad intero mediante il metodo delle stringhe stoi);
* numero di ore di laboratorio del corso contenuto in “interoCorso [6]” (convertito da stringa ad intero mediante il metodo delle stringhe stoi);

Nel caso in cui il primo elemento del vector “interoCorso” (ovvero “interoCorso [0]”) contenga la stringa “a” allora si passa all’elaborazione dei dati inerenti uno specifico anno accademico.

Per migliorare la leggibilità del codice si copia il vector “interoCorso” in un altro chiamato “SpecificYearCourse”. Pertanto tale vector presenta per quello specifico anno accademico le seguenti informazioni:

* “SpecificYearCourse [1]” contiene l’anno accademico del corso (anno inizio – anno fine);
* “SpecificYearCourse [2]” contiene l’info se tale corso è attivo/non attivo;
* “SpecificYearCourse [3]” contiene il numero di versioni in parallelo;
* “SpecificYearCourse [4]” contiene l’elenco degli id dei professori che insegnano quel corso quel particolare anno, compreso il professore titolare, e relative ore di lezione, laboratorio ed esercitazione proprio come riportato nel file;
* “SpecificYearCourse [5]” contiene le info inerenti l’esame di quel corso proprio come sono scritte nel file;
* “SpecificYearCourse [6]” contiene l’elenco dei codici dei corsi raggruppati ovvero dei corsi che si trovano nello stesso anno accademico insieme a quello in esame;

Si salva in una variabile di appoggio denominata “acYear” l’anno accademico contenuto ora in “SpecificYearCourse [1]”; si controlla se quel determinato corso in quell’anno fosse attivo o meno andando a confrontare il contenuto del campo “SpecificYearCourse [2]” con la stringa “attivo”; nel caso in cui lo è viene settata come true la variabile di appoggio “isActive” che tiene appunto conto dello stato del corso; si procede analogamente per il numero di versioni in parallelo contenute nella stringa “SpecificYearCourse [3]” e poi salvate in una variabile intera mediante il metodo di conversione delle stringhe stoi.

Visto il contenuto di “SpecificYearCourse [4]” il quale contiene i diversi dati proprio come scritti nel file (ovvero con caratteri delimitatori come parentesi graffe, quadre e virgole) nasce l’esigenza di togliere le parentesi quadre motivo per cui tale vector viene passato al metodo delle stringhe substr che copia nella stringa “profSenzaQuadre” tutti i caratteri presenti in “SpecificYearCourse [4]” dal secondo al penultimo andando ad escludere così il primo e l’ultimo, rispettivamente “[“ e “]”.

Successivamente si passa il vector “profSenzaQuadre” alla funzione getProfPar che restituisce il vector di stringhe “profCorsoPar” composta da un numero di campi pari al numero di versioni in parallelo dove ogni campo contiene quindi tutti i professori, compreso il titolare, che insegnano quel determinato corso parallelo.

In seguito vengono processate le informazioni riguardanti gli esami contenute in “SpecificYearCourse [5]” copiandole, private delle parentesi graffe aperte e chiuse (mediante il metodo substr) in un vector di stringhe di appoggio detto “examData”. Passando quest’ultimo alla funzione splittedLine insieme al carattere separatore “,” si ottiene il vector “splittedExamData” composto dalle seguenti voci:

* “splittedExamData [0]” contiene la durata dell’esame in minuti;
* “splittedExamData [1]” contiene i minuti necessari per accedere all’esame;
* “splittedExamData [2]” contiene i minuti necessari per abbandonare il luogo dell’esame una volta terminato;
* “splittedExamData [3]” contiene la tipologia di esame “S”/”O” (ovvero scritto/orale);
* “splittedExamData [4]” contiene la tipologia di locazione in cui verrà sostenuto l’esame ”A”/”L” (ovvero aula/laboratorio);

In seguito si prende la stringa contente l’elenco dei codici dei corsi raggruppati contenuti in “SpecificYearCourse [6]” privandoli delle parentesi graffe aperte e chiuse sempre mediante il metodo substr, copiando della stringa tutti i caratteri esclusi il primo (ovvero “{“ ) e l’ultimo (ovvero "}”). La stringa risultante viene passato alla funzione “splittedLine” insieme al carattere separatore che in questo caso è “,”; si ottiene quindi un vector di stringhe “idGrouped”.

QUI MANCA UN PEZZO … vedi messaggi telegram

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo, tenendo conto che possono esserci per uno stesso anche due o più righe che iniziano con la stringa “c” ovvero più anni accademici inerenti quello specifico corso.

Tale metodo si conclude con la chiusura del file database dei corsi di studio “db\_corsi.txt”.

* **addStuds**

prototipo: bool University::addStuds(const std::string &fileIn)

Questo metodo si occupa dell’inserimento dei nuovi studente (quindi privi di matricola) nella mappa in memoria degli studenti. Tali studenti sono riportati nel file il cui nome viene passato nel parametro “fileIn” del metodo stesso.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso (restituendo un booleano pari a false), grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “tokens”. Si controlla che tale vector contenga tre campi ovvero che siano state riportate nella riga per quel determinato studente tutte e tre le info necessarie per salvarlo in memoria ovvero: nome, cognome ed email. In caso in cui il numero di campi dovesse essere diverso da tre viene lanciata una eccezione in cui si specifica il numero corrispondente alla riga in cui si è verificato l’errore (tale variabile contatore che tiene conto del numero di linee lette viene incrementata ogni volta che si rientra nel ciclo del while ed è dichiarata sotto il nome di “line\_counter”).

Nel caso in cui la riga contenga tutte le informazioni necessarie si procede con l’ottenere il codice univoco identificativo per il nuovo studente da inserire mediante l’ausilio dell’apposito metodo “getNewStudentId” creato per la classe “University” (tale metodo viene descritto nel dettaglio nelle righe avvenire).

Ottenuta la matricola dello studente salvata nella variabile “matr”, mediante il metodo insert, si procede con il salvataggio nella mappa in memoria dello studente e delle relative informazioni contenute nei diversi campi del vector “tokens”:

* “tokens [0]” contiene uno o più nomi dello studente (in quanto la funzione splittedLine è stata applicata rispetto al carattere “ ;”;
* “tokens [1]” contiene uno o più cognomi dello studente (in quanto la funzione splittedLine è stata applicata rispetto al carattere “ ;”;
* “tokens [2]” contiene l’indirizzo e-mail dello studente;

La chiave della mappa degli studenti è quindi la matricola dello studente in quanto univoca per ogni studente.

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

Dopo la chiusura del file il metodo si conclude con la riscrittura automatizzata del file database degli studenti db\_studenti.txt, dal momento che nuovi studenti sono stati aggiunti nella base dati, il tutto attraverso il metodo dbStudsWrite (che verrà spiegato solo per puro ordine espositivo nelle righe avvenire).

Tale metodo restituisce, come si può notare dal prototipo, un valore true se giunti sino alla riscrittura del file database.

* **addProfessors**

prototipo: bool University::addProfessors(const std::string &fileIn)

Questo metodo si occupa dell’inserimento dei nuovi professori (quindi privi di matricola) nella mappa in memoria dei professori. Tali professori sono riportati nel file il cui nome viene passato nel parametro “fileIn” del metodo stesso.

Tale metodo si sviluppa in modo analogo al precedente degli studenti.

Si riportano di seguito le informazioni contenute nei diversi campi del vector “tokens”:

* “tokens [0]” contiene uno o più nomi del professore (in quanto la funzione splittedLine è stata applicata rispetto al carattere “ ;”;
* “tokens [1]” contiene uno o più cognomi del professore (in quanto la funzione splittedLine è stata applicata rispetto al carattere “ ;”;
* “tokens [2]” contiene l’indirizzo e-mail del professore;

La chiave della mappa dei professori è quindi la matricola del professore in quanto univoca per ogni prof.

Dopo la chiusura del file il metodo si conclude con la riscrittura automatizzata del file database dei professori db\_professori.txt, dal momento che nuovi professori sono stati aggiunti nella base dati, il tutto attraverso il metodo dbProfsWrite (che verrà spiegato solo per puro ordine espositivo nelle righe avvenire).

Tale metodo restituisce, come si può notare dal prototipo, un valore true se giunti sino alla riscrittura del file database.

* **addClassrooms**

prototipo: bool University::addClassrooms(const std::string &fileIn)

Questo metodo si occupa dell’inserimento di nuove aule (quindi prive di codice identificativo) nella mappa in memoria delle aule. Tali aule sono riportate nel file il cui nome viene passato nel parametro “fileIn” del metodo stesso.

Tale metodo si sviluppa in modo analogo al precedente degli studenti.

Si riportano di seguito le informazioni contenute nei diversi campi del vector “tokens”:

* “tokens [0]” contiene la stringa che specifica il tipo di aula: “A” in caso di aula, “L” in caso di laboratorio;
* “tokens [1]” contiene il nome dell’aula;
* “tokens [2]” contiene la capienza massima dell’aula;
* “tokens [3]” contiene la capienza massima dell’aula se adibita per un esame;

La chiave della mappa delle aule è quindi il codice identificativo in quanto univoco per ogni aula.

Dopo la chiusura del file il metodo si conclude con la riscrittura automatizzata del file database delle aule db\_aule.txt, dal momento che nuove aule sono state aggiunte nella base dati, il tutto attraverso il metodo dbClassRoomWrite (che verrà spiegato solo per puro ordine espositivo nelle righe avvenire).

Tale metodo restituisce, come si può notare dal prototipo, un valore true se giunti sino alla riscrittura del file database.

* **addStudyCourses**

prototipo: bool University::addStudyCourses(const std::string &fin)

Questo metodo si occupa dell’inserimento dei nuovi corsi di studio (quindi privi di codice identificativo) nella mappa in memoria dei corsi di studio. Tali corsi di studio sono riportati nel file il cui nome viene passato nel parametro “fin” del metodo stesso.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso (restituendo in tal caso un booleano pari a false), grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “tokens”. Si controlla che tale vector contenga due campi ovvero che siano state riportate nella riga per quel determinato corso di studio tutte e due le info necessarie per salvarlo in memoria ovvero: il tipo di corso di studio (ovvero le due stringhe che specificano il tipo di corso di studio: BS in caso di laurea triennale, MS in caso di laurea magistrale) e l’elenco dei corsi presenti in tale percorso di studio. In caso in cui il numero di campi dovesse essere diverso da due viene lanciata una eccezione in cui si specifica il numero corrispondente alla riga in cui si è verificato l’errore (tale variabile contatore che tiene conto del numero di linee lette viene incrementata ogni volta che si rientra nel ciclo del while ed è dichiarata sotto il nome di “line\_counter”).

Nel caso in cui la riga contenga tutte le informazioni necessarie si procede con l’ottenere il codice univoco identificativo per il nuovo corso di studio da inserire mediante l’ausilio dell’apposito metodo “getStudyCourseId” creato per la classe “University” (tale metodo viene descritto nel dettaglio nelle righe avvenire).

Ottenuta la matricola del corso di studio salvata nella variabile “codCorso”, visto il valore della stringa “tokens [1]” (contenente l’elenco dei codici identificativi dei diversi corsi raggruppati per semestre all’interno delle parentesi graffe) nasce l’esigenza di separare i diversi corsi per ogni semestre. Si usa il metodo delle stringhe find\_first\_of il quale, relativamente alla stringa “tokens [1]”, passandogli i due caratteri “{“ e “}”, restituisce un intero (salvato nella variabile di appoggio “found”) indicante la prima posizione in corrispondenza della quale si trova nella stringa uno dei due caratteri passati. Dal momento che la find\_first\_of si blocca la primo carattere trovato allora si ripete tale operazione per tutti i rimanenti caratteri di “tokens [1]” successivi a quello appena individuato(motivo per cui nella find\_first\_of è presente come parametro “found+1”) mediante un ciclo while. Le diverse posizioni trovate vengono di volta in volta per ogni ciclo while accodate in un vector di interi detto “posSem” il quale pertanto contiene le diverse posizioni che occupano le parentesi graffe (aperte e chiuse) all’interno di “tokens [1]”, posizione intesa come numero di caratteri che, a partire dal primo presente in “tokens [1]”, si trova la parentesi graffa aperta o chiusa.

A questo punto, grazie ad un ciclo for, per ogni parentesi graffa aperta, che vista la formattazione del file “db\_corsi\_studio.txt“ si trovano nelle posizioni pari del vector “posSem” (motivo per cui l’indice del for viene incrementato di due ad ogni ciclo), si eseguono le seguenti operazioni :

1. si prende la posizione della parentesi graffa aperta, la si salva in una variabile di appoggio “posStart”;
2. si calcola la distanza “len” tra la posizione della “{“ e la posizione della successiva “}”; distanza intesa sempre in termini di caratteri presenti tra le due parentesi;
3. tramite il metodo delle stringhe “substr” si copiano in una stringa tutti i caratteri presenti nella stringa “tokens [1]” che vanno dalla posizione presente in “posStart” (ovvero quella della parentesi graffa aperta) per un numero di caratteri pari al valore contenuto nella variabile “len”; il metodo “substr” restituisce così una stringa contenente i codici di tutti i corsi di un solo semestre; tale stringa viene poi copiata in coda, per ogni ciclo for, nel vector “semestri”.

Alla fine di questo for il vector “semestri” conterrà i corsi di ogni semestre disposti al suo interno in modo che ogni campo di “semestri” contiene tutti i corsi di un certo semestre nel seguente modo:

* “semestri [0]” contiene tutti i corsi del 1o anno 1o semestre (della laurea triennale o magistrale);
* “semestri [1]” contiene tutti i corsi del 1o anno 2o semestre (della laurea triennale o magistrale);
* “semestri [2]” contiene tutti i corsi del 2o anno 1o semestre (della laurea triennale o magistrale);
* “semestri [3]” contiene tutti i corsi del 2o anno 2o semestre (della laurea triennale o magistrale);
* “semestri [4]” contiene tutti i corsi del 3o anno 1o semestre (della laurea triennale);
* “semestri [5]” contiene tutti i corsi del 3o anno 2o semestre (della laurea triennale);

Si controlla a questo punto che, a seconda del tipo laurea, il numero di semestri sia coerente ovvero si controlla che il numero di semestri per una laurea triennale siano sei o che per una laurea triennale siano quattro. Nel caso in cui così non fosse si lancia una eccezione indicando anche la linea in corrispondenza della quale si è verificato l’errore.

Dopo aver letto dalla riga tutte le informazioni inerenti un determinato corso di studio e sistemate in delle apposite variabili si passa alla creazione di un oggetto di appoggio denominato “SCourse” passando al relativo costruttore il codice identificativo del corso di studio e l’informazione inerente il tipo di corso di studio, ovvero percorso triennale o magistrale.

Con l’ausilio del metodo “addSemesterCourses”, appositamente creato per gli oggetti della classe “StudyCourse” (come dimostra il file “StudyCourse.h”), si riempie la mappa privata interna dei semestri dell’oggetto di appoggio poc’anzi creato, “SCourse”, la quale contiene i diversi corsi per ogni di semestre di ogni anno; pertanto mediante un ciclo for, per ogni stringa del vector “semestri” si procede come spiegato di seguito:

1. Si calcola l’anno associato al corso/i presente nella i-esima posizione in “semestri”; siccome il vector “semestri” contiene i corsi di ogni semestre disposti al suo interno in ordine crescente per semestre ovvero partendo dal 1o semestre del 1o anno fino al 2o semestre del 3o (o fino al 2o semestre del 2o anno in caso di laurea magistrale) è facile dedurre come l’anno sia pari al numero del semestre (incrementato di 1 perché la numerazione degli indici in un vector parte da 0) diviso 2 (in quanto ogni anno contiene due semestri);
2. Analogamente si calcola il semestre (se 1o o 2o) associato all’i-esimo corso/i che pertanto risulta essere uguale al resto dell’operazione che ha portato al calcolo dell’anno;
3. Si riempie la mappa privata dell’oggetto “SCourse” grazie al metodo “addSemesterCourses” al quale viene passato: l’anno associato all’i-esimo corso/i, il semestre (1o o 2o) associato all’i-esimo corso/i ed infine l’i-esimo corso/i;

Come ultima operazione si riempie la mappa interna privata denominata “\_studyCourse” della classe “University” (rispettando la convenzione dell’anteporre un underscore al nome dei dati privati di una classe) con il metodo “insert” al quale viene passato come chiave della mappa il codice del corso e come value corrispondente l’oggetto di appoggio “SCourse”.

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

La chiave della mappa dei corsi di studio è quindi il codice identificativo in quanto univoco per ogni corso di studio.

Dopo la chiusura del file il metodo si conclude con la riscrittura automatizzata del file database dei corsi di studio db\_corsi\_studio.txt, dal momento che nuovi corsi di studio sono stati aggiunti nella base dati, il tutto attraverso il metodo dbStudyCourseWrite (che verrà spiegato solo per puro ordine espositivo nelle righe avvenire).

Tale metodo restituisce, come si può notare dal prototipo, un valore true se giunti sino alla riscrittura del file database.

* **addCourses**

prototipo: bool University::addCourses(const std::string &fin)

Questo metodo si occupa dell’inserimento dei nuovi corsi (quindi privi di codice identificativo) nella mappa in memoria dei corsi. Tali corsi sono riportati nel file il cui nome viene passato nel parametro “fin” del metodo stesso.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso (restituendo in tal caso un booleano pari a false), grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “specificYearCourse”. Si controlla che tale vector contenga dieci campi ovvero che siano state riportate nella riga per quel determinato corso di studio tutte e dieci le info necessarie per salvarlo in memoria ovvero:

1. anno accademico;
2. titolo;
3. numero di cfu;
4. numero di ore destinate alle lezioni;
5. numero di ore destinate alle esercitazioni;
6. numero di ore rivolte ai laboratori;
7. numero di versioni in parallelo esistenti per questo corso;
8. elenco degli id dei professori che insegnano quel corso in quel particolare anno, compreso il professore titolare, e relative ore di lezione, laboratorio ed esercitazione;
9. elenco delle info riguardanti l’esame comprendenti la durata dell’esame, il tempo necessario per accedere all’esame, il tempo necessario per abbandonare l’esame, modalità di svolgimento, luogo in cui si terrà;
10. elenco degli id dei corsi raggruppati a quello attualmente in esame;

In caso in cui il numero di campi dovesse essere diverso da dieci viene lanciata una eccezione in cui si specifica il numero corrispondente alla riga in cui si è verificato l’errore (tale variabile contatore che tiene conto del numero di linee lette viene incrementata ogni volta che si rientra nel ciclo del while ed è dichiarata sotto il nome di “line\_counter”).

Nel caso in cui la riga contenga tutte le informazioni necessarie ci si assicura che il corso che viene aggiunto non esista già nel database motivo per cui si procede con un ciclo for nel quale si scandaglia l’intera mappa dei corsi in memoria mediante l’uso di un iteratore che quindi parte dal primo corso in mappa sino all’ultimo (questo possibile grazie all’uso rispettivamente dei metodi begin ed end rivolti agli iteratori). Nel caso in cui dovesse essere presente in memoria un corso uguale a quello che si vuole caricare (appena letto da file), avente pertanto stesso titolo e stesso numero di cfu (caratteristiche che assieme, in assenza di un id, risultano univoche nell’identificazione di un corso) si procede con il lancio di una eccezione che avvisa l’utente della presenza di un corso già esistente informandolo anche del numero della riga corrispondente a quanto accaduto (ancora una volta mediante l’uso di una variabile contatore, “line\_counter”).

Nel caso in cui tale corso non sia già esistente si procede quindi con l’ottenere il codice univoco identificativo per il nuovo corso da inserire mediante l’ausilio dell’apposito metodo “getNewCourseId” creato per la classe “University” (tale metodo viene descritto nel dettaglio nelle righe avvenire).

Si riempie in seguito la mappa interna privata denominata “\_courses” della classe “University” (rispettando la convenzione dell’anteporre un underscore al nome dei dati privati di una classe) con il metodo “insert” al quale viene passato come chiave della mappa il codice del corso (contenuto nella variabile “newIdCourse” e come value corrispondente l’oggetto della classe “Course”, inizializzato mediante costruttore al quale vengono passati (come da definizione presente nel file “Course.h”) i seguenti dati:

1. titolo del corso contenuto in “specificYearCourse [1]”;
2. numero di cfu del corso contenuto in “interoCorso [2]” (convertito da stringa ad intero mediante il metodo delle stringhe stoi);
3. numero delle ore di lezione del corso contenuto in “interoCorso [3]” (convertito da stringa ad intero mediante il metodo delle stringhe stoi);
4. numero delle ore di esercitazione del corso contenuto in “interoCorso [4]” (convertito da stringa ad intero mediante il metodo delle stringhe stoi);
5. numero di ore di laboratorio del corso contenuto in “interoCorso [5]” (convertito da stringa ad intero mediante il metodo delle stringhe stoi);

Si salva in una variabile di appoggio denominata “acYear” l’anno accademico contenuto ora in “SpecificYearCourse [0]”; il numero di versioni in parallelo contenute nella stringa “SpecificYearCourse [6]” vengono salvate in una variabile intera mediante il metodo di conversione delle stringhe stoi.

Visto il contenuto di “SpecificYearCourse [7]” il quale contiene i diversi dati proprio come scritti nel file (ovvero con caratteri delimitatori come parentesi graffe, quadre e virgole) nasce l’esigenza di togliere le parentesi quadre motivo per cui tale vector viene passato al metodo delle stringhe substr che copia nella stringa “profSenzaQuadre” tutti i caratteri presenti in “SpecificYearCourse [7]” dal secondo al penultimo andando ad escludere così il primo e l’ultimo, rispettivamente “[“ e “]”.

Successivamente si passa il vector “profSenzaQuadre” alla funzione getProfPar che restituisce il vector di stringhe “profCorsoPar” composta da un numero di campi pari al numero di versioni in parallelo dove ogni campo contiene quindi tutti i professori, compreso il titolare, che insegnano quel determinato corso parallelo.

In seguito vengono processate le informazioni riguardanti gli esami contenute in “SpecificYearCourse [8]” copiandole, private delle parentesi graffe aperte e chiuse (mediante il metodo substr) in un vector di stringhe di appoggio detto “examData”. Passando quest’ultimo alla funzione splittedLine insieme al carattere separatore “,” si ottiene il vector “splittedExamData” composto dalle seguenti voci:

* “splittedExamData [0]” contiene la durata dell’esame in minuti;
* “splittedExamData [1]” contiene i minuti necessari per accedere all’esame;
* “splittedExamData [2]” contiene i minuti necessari per abbandonare il luogo dell’esame una volta terminato;
* “splittedExamData [3]” contiene la tipologia di esame “S”/”O” (ovvero scritto/orale);
* “splittedExamData [4]” contiene la tipologia di locazione in cui verrà sostenuto l’esame ”A”/”L” (ovvero aula/laboratorio);

In seguito si prende la stringa contente l’elenco dei codici dei corsi raggruppati contenuti in “SpecificYearCourse [9]” privandoli delle parentesi graffe aperte e chiuse sempre mediante il metodo substr, copiando della stringa tutti i caratteri esclusi il primo (ovvero “{“ ) e l’ultimo (ovvero "}”). La stringa risultante viene passato alla funzione “splittedLine” insieme al carattere separatore che in questo caso è “,”; si ottiene quindi un vector di stringhe “idGrouped”.

QUI MANCA STESSO PEZZO DELLA READCOURSE… vedi messaggi telegram

Come ultima operazione si riempie la mappa interna privata denominata “\_courses” della classe “University” (rispettando la convenzione dell’anteporre un underscore al nome dei dati privati di una classe) in particolare con il metodo “at” si recupera il nuovo corso da aggiungere, mediante il suo id ricavato, e si procede con l’inserimento delle seguenti informazioni inerenti quel corso:

* anno accademico;
* l’informazione booleana che mi dice che il corso è attivo (in quanto appena aggiunto);
* numero di corsi paralleli;
* vector con un numero di campi pari al numero di verisoni parallele del seguente corso; ogni campo comprende quindi le diverse informazioni riguardanti il prof titolare e tutti gli altri prof che insegnano quel determinato corso parallelo con relative ore di insegnamento;
* vector con le Informazioni dell’esame;
* vector con i diversi corsi raggruppati;
* ………YY\_SEMESTER………………. INERENTE IL PEZZO CHE MANCA
* ………STUDYCOURSE………………. INERENTE IL PEZZO CHE MANCA

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

La chiave della mappa dei corsi di studio è quindi il codice identificativo in quanto univoco per ogni corso di studio.

Dopo la chiusura del file il metodo si conclude con la riscrittura automatizzata del file database dei corsi di studio db\_corsi\_studio.txt, dal momento che nuovi corsi di studio sono stati aggiunti nella base dati, il tutto attraverso il metodo dbCourseWrite (che verrà spiegato solo per puro ordine espositivo nelle righe avvenire).

Tale metodo restituisce, come si può notare dal prototipo, un valore true se giunti sino alla riscrittura del file database stesso.

* **getNewStudentId**

prototipo: const int University::getNewStudentId() const

Tale metodo, come anticipato nei precedenti, si occupa della generazione di un nuovo id (privato del carattere “s”) rivolto agli studenti da aggiungere nel database.

Inizialmente si controlla se il database degli studenti risulti vuoto o meno; qualora la mappa degli studenti in memoria sia vuota tale metodo restituisce il valore 1 al fine di impedire che venga generata una matricola priva di significato pratico come: “s00000”.

Si dichiara un iteratore “last” con deduzione di tipo (ovvero “auto”) che punta all’ultimo studente della mappa.

Si dichiara una variabile denominata “toReturn” la quale recupera il value associato all’iteratore “last” (ovvero la matricola associata all’ultimo studente) e la si incrementa di 1 in quanto si vuole creare una nuova matricola consecutiva all’ultima già presente in memoria.

Infine si ritorna, come mostra il prototipo di tale metodo, un numero intero, nonché la nuova matricola generata.

Si noti come questo metodo è stato marcato come const al fine di impedire che il risultato restituito da tale metodo, ovvero la nuova matricola generata, venga modificato.

* **getNewProfessorId**

prototipo: const int University::getNewProfessorId() const

Tale metodo, come anticipato nei precedenti, si occupa della generazione di un nuovo id (privato del carattere “d”) rivolto ai professori da aggiungere nel database. Tale metodo si sviluppa in modo analogo a quello degli studenti.

Inizialmente si controlla se il database dei professori risulti vuoto o meno; qualora la mappa dei professori in memoria sia vuota tale metodo restituisce il valore 1 al fine di impedire che venga generata una matricola priva di significato pratico come: “d00000”.

Si dichiara un iteratore “last” con deduzione di tipo (ovvero “auto”) che punta all’ultimo professore della mappa.

Si dichiara una variabile denominata “toReturn” la quale recupera il value associato all’iteratore “last” (ovvero la matricola associata all’ultimo professore) e la si incrementa di 1 in quanto si vuole creare una nuova matricola consecutiva all’ultima già presente in memoria.

Infine si ritorna, come mostra il prototipo di tale metodo, un numero intero, nonché la nuova matricola generata.

Si noti come questo metodo è stato marcato come const al fine di impedire che il risultato restituito da tale metodo, ovvero la nuova matricola generata, venga modificato.

* **getNewProfessorId**

prototipo: const int University::getNewClassroomId() const

Tale metodo, come anticipato nei precedenti, si occupa della generazione di un nuovo id (privato del carattere “A”) rivolto alle aule da aggiungere nel database. Tale metodo si sviluppa in modo analogo a quello degli studenti.

Inizialmente si controlla se il database delle aule risulti vuoto o meno; qualora la mappa delle aule in memoria sia vuota tale metodo restituisce il valore 1 al fine di impedire che venga generata un codice identificativo privo di significato pratico come: “A000”.

Si dichiara un iteratore “last” con deduzione di tipo (ovvero “auto”) che punta all’ultima aula della mappa.

Si dichiara una variabile denominata “toReturn” la quale recupera il value associato all’iteratore “last” (ovvero l’id associato all’ultima aula) e la si incrementa di 1 in quanto si vuole creare una nuova matricola consecutiva all’ultima già presente in memoria.

Infine si ritorna, come mostra il prototipo di tale metodo, un numero intero, nonché la nuova matricola generata.

Si noti come questo metodo è stato marcato come const al fine di impedire che il risultato restituito da tale metodo, ovvero la nuova matricola generata, venga modificato.

* **getNewStudyCourseId**

prototipo: const int University::getNewStudyCourseId() const

Tale metodo, come anticipato nei precedenti, si occupa della generazione di un nuovo id (privato del carattere “C”) rivolto ai corsi di studio da aggiungere nel database. Tale metodo si sviluppa in modo analogo a quello degli studenti.

Inizialmente si controlla se il database dei corsi di studio risulti vuoto o meno; qualora la mappa dei corsi di studio in memoria sia vuota tale metodo restituisce il valore 1 al fine di impedire che venga generata un codice identificativo privo di significato pratico come: “C000”.

Si dichiara un iteratore “last” con deduzione di tipo (ovvero “auto”) che punta all’ultimo corso di studio della mappa.

Si dichiara una variabile denominata “toReturn” la quale recupera la chiave associata all’iteratore “last” (ovvero l’id associato all’ultimo corso di studi) e la si incrementa di 1 in quanto si vuole creare una nuova matricola consecutiva all’ultima già presente in memoria.

Infine si ritorna, come mostra il prototipo di tale metodo, un numero intero, nonché la nuova matricola generata.

Si noti come questo metodo è stato marcato come const al fine di impedire che il risultato restituito da tale metodo, ovvero la nuova matricola generata, venga modificato.

* **getNewCourseId**

prototipo: const int University::getNewCourseId() const

Tale metodo, come anticipato nei precedenti, si occupa della generazione di un nuovo codice identificativo rivolto ai corsi da aggiungere nel database. Tale metodo si sviluppa in modo analogo a quello degli studenti.

Si controlla inizialmente se la mappa dei corsi in memoria è vuota o meno; in tal caso l’id da restituire coincide con il primo che si può avere ovvero “01AAAAA”.

Si prende l’ultimo corso presente in memoria (avente quindi l’ultimo id) e lo si salva in un oggetto stringstream detto “Id”. Successivamente si estraggono da quest’ultimo i due campi di cui è composto ovvero una parte numerica detta “num”, costituita da un numero a due cifre, e una parte alfabetica detta “cod” composta da quattro cifre.

Mediante un ciclo for si analizza l’ultima parte dell’id ovvero quella alfabetica iniziando dall’ultimo carattere spostandosi così in ogni ciclo ai caratteri precedenti (in quanto come ogni somma anche quella che opera con dei caratteri prevede di partire dalle posizioni meno significative).

Partendo dall’ultimo carattere si controlla se questo risulta diverso dal carattere “Z”:

* se risulta diverso da “Z” allora si incrementa di uno il carattere in questione, ovvero si passa al carattere successivo in ordine alfabetico e si setta a true un flag booleano detto “noZ” che appena diventa vero provoca al prossimo ciclo l’uscita dal for (in quanto nella condizione del for tale flag negato è in and con l’indice della stringa che deve essere maggiore o uguale a zero, dal momento che l’analisi della stringa è fatta dal fondo verso la testa);
* se risulta uguale a “Z” allora si setta il carattere in questione con quello successivo nell’alfabeto ovvero “A” e si passa con il prossimo ciclo for all’analisi del carattere precedente; pertanto si riparte dal confronto con il carattere “Z”.

Solo nel momento in cui tutti i caratteri della stringa sono pari ad “A” si prosegue con l’incremento di uno della parte numerica, questo viene fatto con un if esterno al for, diversamente si prosegue col conservare il valore vecchio.

Successivamente mediante il metodo setw si decide di scrivere la parte numerica sempre su due cifre anteponendo uno 0 mediante il metodo setfill nel caso in cui la variabile “cod” sia costituita da un numero ad una cifra.

Si copia l’oggetto stringstream in una stringa la quale conterrà pertanto il nuovo codice identifcativo che verrà restituito come mostrato dal prototipo del metodo.

* **updateStuds**

prototipo: bool University::updateStuds(const std::string &fin)

Tale metodo consente di aggiornare le informazioni riguardanti degli studenti già presenti in anagrafica con i nuovi dati riportati in un file apposito, il cui nome viene passato nel parametro “fin” del metodo stesso.

Si ricorda che:

* al fine di poter aggiornare i dati relativi ad un determinato studente risulta obbligatoria la presenza della sua matricola all’interno delle righe del suddetto file;
* in questo file viene rispettato il medesimo formato già usato nel file db\_studenti.txt dove pertanto il carattere separatore è nuovamente “;” dando la possibilità di poter lasciare vuoti alcuni campi, sottintendendo in tal caso il voler conservare i dati già in essere nel database.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “infoStud”.

Successivamente si controlla che il primo campo del vector “infoStud”, che dovrebbe contenere la matricola dello studente (la cui presenza come spiegato prima deve essere obbligatoria), sia vuoto o meno. Qualora lo sia viene pertanto lanciata una eccezione riportando anche la linea, mediante la variabile contatore “line\_counter” in corrispondenza della quale si è verificato tale errore.

Nel caso in cui la matricola dello studente è stata specificata si procede coll’inizializzare un oggetto stringstream detto “ss” con la stringa contenuta in “infoStud[0]” al fine di separare successivamente il carattere “c” dal numero che costituisce la matricola stessa.

In seguito mediante il metodo find, a cui viene passata la matricola dello studente (in quanto chiave univoca di tale mappa) si procede con il ricercare se tale studente sia presente o meno in anagrafica lanciando una eccezione in caso di mancata presenza.

Nel caso in cui lo studente risulti presente in memoria allora si passa con il processare tutte le sue nuove informazioni presenti ognuna nei diversi campi di “infoStud” quindi grazie ad un for ci si muove tra i suoi diversi campi. Dal momento che è lasciata la possibilità di poter conservare le informazioni già presenti allora si controlla che ogni campo pertanto non sia vuoto. A seconda quindi del valore che assume l’indice “i” si passa da un campo all’altro del vector e mediante l’ausilio di uno switch, in base al valore assunto da “i”, si sceglie quale informazione dello studente si vuole aggiornare ovvero:

* siccome infoStud [1] potrebbe contenere il nuovo nome dello studente allora quando i = 1 significa che si sta puntando a tale campo pertanto si passa all’aggiornare il nome dello studente (si è usato il condizionale dal momento che non si ha la certezza che tale campo verrà aggiornato o meno).

Nel caso in cui il nuovo nome risulta diverso dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo nome, “infoStud [1]”, al metodo updateName, creato appositamente per la classe “Student”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Student” che si vuole aggiornare;

* siccome infoStud [2] potrebbe contenere il nuovo cognome dello studente allora quando i = 2 significa che si sta puntando a tale campo pertanto si passa all’aggiornare il cognome dello studente (si è usato il condizionale dal momento che non si ha la certezza che tale campo verrà aggiornato o meno).

Nel caso in cui il nuovo cognome risulta diverso dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo cognome, “infoStud [2]”, al metodo updateSurName, creato appositamente per la classe “Student”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Student” che si vuole aggiornare;

* siccome infoStud [3] potrebbe contenere la nuova email dello studente allora quando i = 3 significa che si sta puntando a tale campo pertanto si passa all’aggiornare la mail dello studente (si è usato il condizionale dal momento che non si ha la certezza che tale campo verrà aggiornato o meno).

Nel caso in cui la nuova mail risulta diverso dalla precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio della nuova email, “infoStud [3]”, al metodo updateEmail, creato appositamente per la classe “Student”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Student” che si vuole aggiornare;

Il ciclo while si conclude con incrementare di 1 la variabile contatore line\_counter.

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

dal momento che diverse informazioni riguardanti alcuni studenti sono state aggiornate allora dopo la chiusura del file il metodo si conclude con la riscrittura automatizzata del file database degli studenti db\_studenti.txt il tutto attraverso il metodo dbStudsWrite (che verrà spiegato solo per puro ordine espositivo nelle righe avvenire).

Tale metodo, updateStuds, restituisce, come si può notare dal prototipo, un valore true se giunti sino alla riscrittura del file database.

* **updateProfessors**

prototipo: bool University::updateProfessors(const std::string &fin)

Tale metodo consente di aggiornare le informazioni riguardanti dei professori già presenti in anagrafica con i nuovi dati riportati in un file apposito, il cui nome viene passato nel parametro “fin” del metodo stesso.

Si ricorda che:

* al fine di poter aggiornare i dati relativi ad un determinato professore risulta obbligatoria la presenza della sua matricola all’interno delle righe del suddetto file;
* in questo file viene rispettato il medesimo formato già usato nel file db\_professori.txt dove pertanto il carattere separatore è nuovamente “;” dando la possibilità di poter lasciare vuoti alcuni campi, sottintendendo in tal caso il voler conservare i dati già in essere nel database.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “infoProf”.

Successivamente si controlla che il primo campo del vector “infoProf”, che dovrebbe contenere la matricola del professore (la cui presenza come spiegato prima deve essere obbligatoria), sia vuoto o meno. Qualora lo sia viene pertanto lanciata una eccezione riportando anche la linea, mediante la variabile contatore “line\_counter” in corrispondenza della quale si è verificato tale errore.

Nel caso in cui la matricola del professore è stata specificata si procede coll’inizializzare un oggetto stringstream detto “ss” con la stringa contenuta in “infoProf[0]” al fine di separare successivamente il carattere “d” dal numero che costituisce la matricola stessa.

In seguito mediante il metodo find, a cui viene passata la matricola del professore (in quanto chiave univoca di tale mappa) si procede con il ricercare se tale professore sia presente o meno in anagrafica lanciando una eccezione in caso di mancata presenza.

Nel caso in cui il professore risulti presente in memoria allora si passa con il processare tutte le sue nuove informazioni presenti ognuna nei diversi campi di “infoProf” quindi grazie ad un for ci si muove tra i suoi diversi campi. Dal momento che è lasciata la possibilità di poter conservare le informazioni già presenti allora si controlla che ogni campo pertanto non sia vuoto. A seconda quindi del valore che assume l’indice “i” si passa da un campo all’altro del vector e mediante l’ausilio di uno switch, in base al valore assunto da “i”, si sceglie quale informazione del professore si vuole aggiornare ovvero:

* siccome infoProf [1] potrebbe contenere il nuovo nome del professore allora quando i = 1 significa che si sta puntando a tale campo pertanto si passa all’aggiornare il nome del professore (si è usato il condizionale dal momento che non si ha la certezza che tale campo verrà aggiornato o meno).

Nel caso in cui il nuovo nome risulta diverso dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo nome, “infoProf [1]”, al metodo updateName, creato appositamente per la classe “Professor”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Professor” che si vuole aggiornare;

* siccome infoProf [2] potrebbe contenere il nuovo cognome del professore allora quando i = 2 significa che si sta puntando a tale campo pertanto si passa all’aggiornare il cognome del professore (si è usato il condizionale dal momento che non si ha la certezza che tale campo verrà aggiornato o meno).

Nel caso in cui il nuovo cognome risulta diverso dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo cognome, “infoProf [2]”, al metodo updateSurName, creato appositamente per la classe “Professor”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Professor” che si vuole aggiornare;

* siccome infoProf [3] potrebbe contenere la nuova email del professore allora quando i = 3 significa che si sta puntando a tale campo pertanto si passa all’aggiornare la mail del professore (si è usato il condizionale dal momento che non si ha la certezza che tale campo verrà aggiornato o meno).

Nel caso in cui la nuova mail risulta diverso dalla precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio della nuova email, “infoProf [3]”, al metodo updateEmail, creato appositamente per la classe “Professor”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Professor” che si vuole aggiornare;

Il ciclo while si conclude con incrementare di 1 la variabile contatore line\_counter.

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

dal momento che diverse informazioni riguardanti alcuni professori sono state aggiornate allora dopo la chiusura del file il metodo si conclude con la riscrittura automatizzata del file database dei professori db\_professori.txt il tutto attraverso il metodo dbProfsWrite (che verrà spiegato solo per puro ordine espositivo nelle righe avvenire).

Tale metodo, updateProfs, restituisce, come si può notare dal prototipo, un valore true se giunti sino alla riscrittura del file database.

* **updateClassroom**

prototipo: bool University::updateClassroom(const std::string &fin)

Tale metodo consente di aggiornare le informazioni riguardanti le aule già presenti in anagrafica con i nuovi dati riportati in un file apposito, il cui nome viene passato nel parametro “fin” del metodo stesso.

Si ricorda che:

* al fine di poter aggiornare i dati relativi ad una determinata aula risulta obbligatoria la presenza della sua matricola all’interno delle righe del suddetto file;
* in questo file viene rispettato il medesimo formato già usato nel file db\_aule.txt dove pertanto il carattere separatore è nuovamente “;” dando la possibilità di poter lasciare vuoti alcuni campi, sottintendendo in tal caso il voler conservare i dati già in essere nel database.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “infoClassroom”.

Successivamente si controlla che il primo campo del vector “infoClassroom”, che dovrebbe contenere la matricola dell’aula (la cui presenza come spiegato prima deve essere obbligatoria), sia vuoto o meno. Qualora lo sia viene pertanto lanciata una eccezione riportando anche la linea, mediante la variabile contatore “line\_counter” in corrispondenza della quale si è verificato tale errore.

Nel caso in cui la matricola dell’aula sia stata specificata si procede coll’inizializzare un oggetto stringstream detto “ss” con la stringa contenuta in “infoClassroom [0]” al fine di separare successivamente il carattere “A” dal numero che costituisce la matricola stessa.

In seguito mediante il metodo find, a cui viene passata la matricola dell’aula (in quanto chiave univoca di tale mappa) si procede con il ricercare se tale aula sia presente o meno in anagrafica lanciando una eccezione in caso di mancata presenza.

Nel caso in cui l’aula risulti presente in memoria allora si passa con il processare tutte le sue nuove informazioni presenti ognuna nei diversi campi di “infoClassroom” quindi grazie ad un for ci si muove tra i suoi diversi campi. Dal momento che è lasciata la possibilità di poter conservare le informazioni già presenti allora si controlla che ogni campo pertanto non sia vuoto. A seconda quindi del valore che assume l’indice “i” si passa da un campo all’altro del vector e mediante l’ausilio di uno switch, in base al valore assunto da “i”, si sceglie quale informazione dell’aula si vuole aggiornare ovvero:

* siccome infoClassroom [1] potrebbe contenere il tipo di aula (aula o laboratorio) allora quando i = 1 significa che si sta puntando a tale campo pertanto si passa all’aggiornare il tipo di aula (si è usato il condizionale dal momento che non si ha la certezza che tale campo verrà aggiornato o meno).

Essendo un campo di tipo char allora nel caso in cui “infoClassroom [1]” contenga il carattere “L” allora si setta al valore true una variabile booleana di appoggio, detta “lab”; in caso contrario ovvero se “infoClassroom [1]” contiene il carattere “A” si procede con il settare “lab” al valore false.

Nel caso in cui il nuovo tipo risulti diverso dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo tipo, il cui stato booleano è contenuto in “lab”, al metodo updateClassroom, creato appositamente per la classe “Classroom”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Classroom” che si vuole aggiornare;

* siccome infoClassroom [2] potrebbe contenere il nuovo nome dell’aula allora quando i = 2 significa che si sta puntando a tale campo pertanto si passa all’aggiornare il nome dell’aula (si è usato il condizionale dal momento che non si ha la certezza che tale campo verrà aggiornato o meno).

Nel caso in cui il nuovo nome risulta diverso dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo nome, “infoClassroom [2]”, al metodo updateName, creato appositamente per la classe “Classroom”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Classroom” che si vuole aggiornare;

* siccome infoClassroom [3] potrebbe contenere il numero di posti a sedere dell’aula allora quando i = 3 significa che si sta puntando a tale campo pertanto si passa all’aggiornare il numero di posti a sedere dell’aula (si è usato il condizionale dal momento che non si ha la certezza che tale campo verrà aggiornato o meno).

Nel caso in cui il numero di posti a sedere dell’aula risulta diverso dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo valore, “infoClassroom [3]”, al metodo updateNSeats, creato appositamente per la classe “Classroom”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Classroom” che si vuole aggiornare;

* siccome infoClassroom [4] potrebbe contenere il numero di posti a sedere dell’aula durante un esame allora quando i = 3 significa che si sta puntando a tale campo pertanto si passa all’aggiornare il numero di posti a sedere dell’aula durante un esame (si è usato il condizionale dal momento che non si ha la certezza che tale campo verrà aggiornato o meno).

Nel caso in cui il numero di posti a sedere dell’aula in caso di esame risulta diverso dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo valore, “infoClassroom [4]”, al metodo updateNExamSeats, creato appositamente per la classe “Classroom”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Classroom” che si vuole aggiornare;

Il ciclo while si conclude con incrementare di 1 la variabile contatore line\_counter.

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

dal momento che diverse informazioni riguardanti alcune aule sono state aggiornate allora dopo la chiusura del file il metodo si conclude con la riscrittura automatizzata del file database dele aule db\_aule.txt il tutto attraverso il metodo dbClassRoomWrite (che verrà spiegato solo per puro ordine espositivo nelle righe avvenire). Tale metodo, updateClassroom, restituisce, come si può notare dal prototipo, un valore true se giunti sino alla riscrittura del file database.

* **insertCourses**

prototipo: bool University::insertCourses(const std::string &fin)

Tale metodo consente di aggiornare le informazioni riguardanti un determinato corso già presente in anagrafica con i nuovi dati riportati in un file apposito, il cui nome viene passato nel parametro “fin” del metodo stesso.

Si ricorda che:

* al fine di poter aggiornare i dati relativi ad un determinato corso risulta obbligatoria la presenza della sua matricola all’interno delle righe del suddetto file;
* in questo file viene rispettato il medesimo formato già usato nel file db\_corsi.txt dove pertanto il carattere separatore è nuovamente “;” dando la possibilità di poter lasciare vuoti alcuni campi, sottintendendo in tal caso il voler conservare i dati già in essere nel database relativi all’ano accdemico precedente.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “specificYearCourse”.

Successivamente si controlla se tale corso risulta già presente in memoria il tutto mediante una find applicata al primo campo del vector, “specificYearCourse” [0] in quanto quest’ultimo contine l’id univoco del corso stesso. Nel caso in cui tale corso non risulta presente allora la find restituisce l’iteratore all’ultimo elemento e in tal caso viene pertanto lanciata una eccezione che spiega l’assenza di tale corso.

Nel caso in cui sia presente invece si procede coll’inizializzare un iteratore, “course\_with\_given\_id”, con quello già in memoria e corrispondente quindi a ““specificYearCourse” [0]”.

Si controlla se il corso dell’anno precedente contiene tutte le informazioni il tutto mediante il metodo courseOfTheYearIsEmpty applicato all’oggetto corso dell’anno precedente.

Solo nel caso in cui tutti i campi dell’anno precedente non sono vuoti si passa con il riempire, mediante il metodo fillSpecificYearCourse, gli eventuali campi vuoti sul corso dell’anno attuale usando le informazioni inerenti lo stesso corso ma dell’anno precedente infatti tale metodo viene applicato all’oggetto corso dell’anno precedente (ovvero quello già presente in memoria) passando come parametro il corso dell’anno attuale.

Qualora vi sia almeno un campo vuoti in merito all’anno precedente allora si passa con il lanciare delle eccezioni al fine di rendere l’utente al corrente di quanto accaduto. In particolare se il corso dell’anno precedente non contiene tutti i campi, di numero sette, allora si lancia una relativa eccezione. Se sono tutti presenti invece l’assenza di info precedentemente individuata potrebbe essere dovuta alla presenza di campi vuoti per questo ci si avvale dell’uso di un for con il quale si controlla campo per campo quale di questi sia appunto vuoto e lanciando in tal caso una eccezione.

Superati questi controlli si passa con l’analizzare la linea appena letta.

Si salva in una variabile di appoggio, “acYear”, l’anno accademico del corso letto da file che corrisponde al secondo campo di “specificYearCourse”, ovvero “specificYearCourse [1]”.

Dal momento che un corso dura un anno accademico ma occupa due anni solari si passa col salvare il primo dei due, ovvero l’anno di inizio, in una variabile “acStartYear” avvalendosi del metodo delle stringhe substr.

Dai diversi campi del vector “specificYearCourse” estraggo le diverse info:

* “specificYearCourse [3]” contiene il numero di corsi in parallelo che si tengono per quello stesso corso e pertanto viene salvato questo dato come intero nella variabile “num\_parallel\_courses”;
* “specificYearCourse [4]” contiene l’elenco dei professori che insegnano tale corso ciascuno con la sua matricola e le sue ore, incluso il prof titolare per ogni versione in parallelo; quindi proprio come accadeva nella read\_Course si analizzano tale info ottenendo il vector “profCorsoPar” che contiene pe ogni suo campo tutti gli id dei prof che appunto insegnano tale corso;
* “specificYearCourse [5]” contiene le info riguardanti l’esame pertanto si salva nel vector “splittedExamData” le diverse info sull’esame come durata dell’esame, tempo necessario per accedere all’esame e per uscire, modalità di sostenimento dell’esame, scritto o orale, e relativo luogo dedicato allo svolgimento.
* “specificYearCourse [6]” contiene gli id di tutti i corsi ad esso raggruppati che vengono salvati separatamente nel vector “idGrouped”;

Successivamente mediante l’ausilio di un for si scandagliano tutti i corsi di studio salvati n memoria e per ognuno di questi si controlla se sia presente o meno il corso che si sta analizzando; in particolare si utilizza un apposito metodo detto isInWhichSemester il quale a sua volta controlla per ogni semestre di quel dato corso di studio è presente il corso in analisi restituendo in caso affermativo l’anno e il semestre in cui compare salvandolo in una apposita variabile di appoggio detta “res”.

Nel caso in cui è stato trovato in vector d’appoggio di interi detto “studyCourse” si salvano gli id dei corsi di studio in cui tale corso compare accodandoli di volta in volta nel vector.

SI controlla inoltre con il metodo controlGroupedCourses che i corsi raggruppati a quello in analisi non appartengano al suo medesimo corso di studio, sollevando in tal caso una eccezione.

Si precisa che tali operazioni appena spiegate risulteranno poi utili per la parte successiva del progetto ovvero la generazione esami.

Successivamente controllo se tale corso risulta attivo o meno, info contenuta in “specificYearCourse [2]” in particolare nel caso in cui risulta attivo setto al valore true un variabile di appoggio booleana detta “isActive”.

Diversamente viene impostata uguale al valore false ricordando che se un corso viene aggiornato diventando spento, ovvero non più attivo, occorre toglierlo dai corsi attivi e aggiungerlo ai corsi spenti procedura che viene messa in atto mediante il for; mediante tale for in particolare si vanno a scandagliare tutti i corsi di studio presenti in memoria e per ognuno di questi che contiene tale corso, mediante il metodo updateSemestersAndOffCourses, si procede col togliere il corso da quelli attivi e spostarlo in quelli spenti.

**Si presti attenzione al fatto che quando un corso diventa spento tale metodo, updateSemestersAndOffCourses, non solo sposta il corso da quelli attivi in quelli spenti, \_offCourses, ma elimina anche ogni sua informazione dalla mappa dei corsi; dal momento che la lista \_offCourses contiene solo i codici dei corsi spenti se in seguito si dovesse presentare la necessità di elaborare una sessione di esami inerente un periodo accademico in cui tale corso era ancora attivo risulta evidente la necessità di reperire le informazioni relative a quel corso che \_offCourse non contiene e che sono state eliminate dal metodo updateSemestersAndOffCourses; pertanto sempre nel metodo updateSemestersAndOffCourses si procede col salvare il codice del corso spento, l’anno accademico in cui è stato spento ed inoltre l’anno e il semestre di appartenenza il tutto nel vector “\_tempInfoNotActiveCoursesToWriteInTheDB”; in particolare ogni campo del vector “\_tempInfoNotActiveCoursesToWriteInTheDB” contiene queste tre info relative ad uno stesso corso spento.**

Infine si aggiorna il corso in memoria relativo all’anno accademico attuale usando le info salvate nelle diverse variabili di appoggio.

Si ricorda che tali operazioni vengono fatte per ogni riga del file perché presenti nel while sopra citato. Come di buona norma si chiude il file aperto all’inizio del metodo mediante l’apposito metodo close.

Successivamente per ogni corso in memoria si effettuano i seguenti controlli/riempimenti:

* Il metodo fillAcYearEmpty controlla per ogni accademico se per quel dato corso mancano degli anni accademici ovvero se c'è un gap di almeno due anni tra la prima testimonianza di corso individuata in memoria e l'ultima. In tal caso tutti gli anni accademici assenti vengono inizializzati con le stesse info del corso relativo all'ultimo anno accademico prima del gap,
* Il metodo notActive controlla per ogni anno accademico se un dato corso, nei diversi anni, è passato da non attivo ad attivo sollevando in tal caso una eccezione in quanto un corso se spento non può essere "riattivato";
* Il metodo sameSemester controlla se i diversi corsi raggruppati a quello che si sta analizzando appartengono allo stesso semestre scatenando una eccezione nel caso in cui non lo siano;
* Il metodo controlReciprocyGrouped controlla che ci sia reciprocità tra corsi raggruppati ovvero se un corso risulta raggruppato con un altro allora quest’ultimo deve risultare raggruppato con il primo;

Solo dopo aver ultimato i controlli si procede con l’aggiornamento dei file database mediante appositi metodi di scrittura su file.

Se giunto sino a questo punto il metodo è stato eseguito senza interruzioni pertanto viene restituito un bool di valore true come dimostra lo stesso prototipo del metodo in esame.

* **dbStudsWrite**

prototipo: void University::dbStudsWrite()

Tale metodo si occupa della riscrittura del file database degli studenti; si dichiara quindi un file detto “fout” che, mediante il metodo open, apre il file database di nome “db\_studenti.txt” in entrambe le seguenti modalità:

* out affinché si possa scrivere su tale file;
* trunc affinché eventuali contenuti già presenti nel file vengano sovrascritti;

Mediante l’ausilio di un for e di un iteratore che punta ad oggetti della classe “Student” si procede col scandagliare tutta la mappa di studenti in memoria. Per ogni ciclo for ognuno di essi viene salvato momentaneamente in un oggetto di appoggio detto “stud” e grazie all’overload dell’operatore “<<” si procede con la scrittura su file “fout” dell’oggetto “stud”; in particolare l’overload dell’operatore “<<” è stato realizzato affinché in fase di scrittura venga rispettata la formattazione richiesta del file “db\_studenti.txt”.

Come buona norma di programmazione il metodo si conclude con la chiusura del file usando l’apposito metodo close.

* **dbProfsWrite**

prototipo: void University::dbProfsWrite()

Tale metodo si occupa della riscrittura del file database dei professori; si dichiara quindi un file detto “fout” che, mediante il metodo open, apre il file database di nome “db\_professorti.txt” in entrambe le seguenti modalità:

* out affinché si possa scrivere su tale file;
* trunc affinché eventuali contenuti già presenti nel file vengano sovrascritti;

Mediante l’ausilio di un for e di un iteratore che punta ad oggetti della classe “Professor” si procede col scandagliare tutta la mappa dei professori in memoria. Per ogni ciclo for ognuno di essi viene salvato momentaneamente in un oggetto di appoggio detto “prof” e grazie all’overload dell’operatore “<<” si procede con la scrittura su file “fout” dell’oggetto “prof”; in particolare l’overload dell’operatore “<<” è stato realizzato affinché in fase di scrittura venga rispettata la formattazione richiesta del file “db\_professori.txt”.

Come buona norma di programmazione il metodo si conclude con la chiusura del file usando l’apposito metodo close.

* **dbClassRoomWrite**

prototipo: void University::dbClassRoomWrite()

Tale metodo si occupa della riscrittura del file database delle aule; si dichiara quindi un file detto “fout” che, mediante il metodo open, apre il file database di nome “db\_aule.txt” in entrambe le seguenti modalità:

* out affinché si possa scrivere su tale file;
* trunc affinché eventuali contenuti già presenti nel file vengano sovrascritti;

Mediante l’ausilio di un for e di un iteratore che punta ad oggetti della classe “Classroom” si procede col scandagliare tutta la mappa delle aule in memoria. Per ogni ciclo for ognuno di essi viene salvato momentaneamente in un oggetto di appoggio detto “room” e grazie all’overload dell’operatore “<<” si procede con la scrittura su file “fout” dell’oggetto “room”; in particolare l’overload dell’operatore “<<” è stato realizzato affinché in fase di scrittura venga rispettata la formattazione richiesta del file “db\_aule.txt”.

Come buona norma di programmazione il metodo si conclude con la chiusura del file usando l’apposito metodo close.

* **dbStudyCourseWrite**

prototipo: void University::dbStudyCourseWrite()

Tale metodo si occupa della riscrittura del file database dei corsi di studio; si dichiara quindi un file detto “fout” che, mediante il metodo open, apre il file database di nome “db\_corsi\_studio.txt” in entrambe le seguenti modalità:

* out affinché si possa scrivere su tale file;
* trunc affinché eventuali contenuti già presenti nel file vengano sovrascritti;

Mediante l’ausilio di un for e di un iteratore che punta ad oggetti della classe “StudyCourse” si procede col scandagliare tutta la mappa dei corsi di studio in memoria. Per ogni ciclo for ognuno di essi viene salvato momentaneamente in un oggetto di appoggio detto “sC” e grazie all’overload dell’operatore “<<” si procede con la scrittura su file “fout” dell’oggetto “sC”; in particolare l’overload dell’operatore “<<” è stato realizzato affinché in fase di scrittura venga rispettata la formattazione richiesta del file “db\_ corsi\_studio.txt”.

Come buona norma di programmazione il metodo si conclude con la chiusura del file usando l’apposito metodo close.

* **dbCourseWrite**

prototipo: void University::dbCourseWrite()

Tale metodo si occupa della riscrittura del file database dei corsi; si dichiara quindi un file detto “fout” che, mediante il metodo open, apre il file database di nome “db\_corsi.txt” in entrambe le seguenti modalità:

* out affinché si possa scrivere su tale file;
* trunc affinché eventuali contenuti già presenti nel file vengano sovrascritti;

Mediante l’ausilio di un for e di un iteratore che punta ad oggetti della classe “Course” si procede col scandagliare tutta la mappa dei corsi in memoria. Per ogni ciclo for ognuno di essi viene salvato momentaneamente in un oggetto di appoggio detto “generalCourse” e grazie all’overload dell’operatore “<<” si procede con la scrittura su file “fout” dell’oggetto “generalCourse”; in particolare l’overload dell’operatore “<<” è stato realizzato affinché in fase di scrittura venga rispettata la formattazione richiesta del file “db\_ corsi.txt”.

Come buona norma di programmazione il metodo si conclude con la chiusura del file usando l’apposito metodo close.

* **controlDatabase**

prototipo: bool University::controlDatabase(int startAcYear)

Tale metodo controlla se i file sono coerenti all’anno accademico che gli viene passato come parametro intero.

Inizialmente si controlla che i database non siano vuoti, operazione che viene attribuita al metodo dataBaseIsEmpty che per una semplice chiarezza espositiva verrà spiegato nelle righe avvenire.

Terminato il metodo, se non è stata sollevata alcuna eccezione da esso, si procede col controllare se un certo corso contenga effettivamente le info relative all’anno accademico richiesto, mediante il metodo controlExistenceSpecificYear appositamente creato. Questa operazione viene ripetuta per ogni corso presente in memoria grazie ad un for. In particolare al metodo controlExistenceSpecificYear, rivolto quindi agli oggetti della classe “Course”, viene passato il codice identificativo del corso e l’anno accademico di interesse.

Nello specifico controlExistenceSpecificYear, usando il metodo count, controlla se nella mappa “\_courseOfTheYear” (ovvero la mappa contenente le info di quel corso per i diversi anni accademici) sono presenti informazioni in merito all’anno passato come parametro al metodo. In caso negativo si procede con lo scatenare una eccezione diversamente viene restituito il valore bool true.

Concluso il for se no sono state sollevate eccezioni il metodo in esame, controlDatabase, restituisce true.

* **dataBaseIsEmpty**

prototipo: bool University::dataBaseIsEmpty(int startAcYear)

Tale metodo come anticipato precedentemente si occupa di controllare se i database in memoria sono vuoti io meno. Tale controllo viene eseguito per ogni database del quale il progetto fa uso scatenando in caso di database vuoto una eccezione specifica. Tale controllo viene quindi rivolto ai seguenti database in memoria:

* \_professors;
* \_students;
* \_courses;
* \_studyCourse;
* \_acYearSessions al quale si controlla se è vuoto ma anche se non sono presenti info sulle sessioni di esami relative all’anno accademico in corso generando in entrambi i casi una eccezione;

Infine tale metodo mediante un for controlla che per ogni corso per quel determinato anno accademico ci siano studenti iscritti e qualora il numero degli iscritti sia uguale a 0 viene sollevata una eccezione specificando il corso che non presenta studenti.

* **controlGroupedCourses**

prototipo: bool University::controlGroupedCourses(int idStudyCourse , std::vector<std::string> &idGrouped, std::string nameCourse , int line\_counter, std::string idCourse )

Il metodo seguente nasce col fine di controllare se i corsi raggruppati contenuti nel vector di stringhe “idGrouped”, passato come parametro, non siano appartenenti al corso di studio identificato dall’id contenuto nel parametro “idStudyCourse” e al quale appartiene il corso passato come parametro mediante “nameCourse” e “idCourse”.

Si salva in un vector di stringhe detto “allCoursesOfCdS” tutti i corsi del corso di studio identificato dal parametro intero “idStudyCourse”. Con l’aiuto di un for, per ogni stringa del vector “idGrouped” si controlla se tale stringa, ovvero se tale corso raggruppato, sia presente tra i corsi del CdS in esame (alias corso di studio in esame). In caso affermativo si solleva una eccezione; con quest’ultima si rende l’utente partecipe del fatto che

il corso raggruppato j-esimo, ovvero “idGrouped [j]”, appartiene allo stesso CdS (indicato da “idCourse”) al quale appartiene il corso “idCourse”.

Il metodo coerentemente con il suo tipo di ritorno restituisce un bool di valore true perché giunti sino al termine del for.

* **thereIsAHoleInTheCoursesCodes**

prototipo: void University::thereIsAHoleInTheCoursesCodes()

Questo metodo controlla che non ci siano buchi negli id dei corsi per tutti i corsi di studio ovvero se tutti i corsi hanno id che sono consecutivi.

Mediante un for, per ogni corso di studio si salvano tutti i suoi corsi in uno stesso campo del vector di stringhe “codesReturned”. In ogni ciclo for si procede col ricopiare ………

DUBBIO André ma qui se il for fa un numero di cicli dispari allora allcoursedcodefronsemester contiene i corsi di codesrereturned nell'ordine opposto mentre se sono pari nello stesso ordine. Non si dovrebbe mettere fuori dal for?

* **checkDistance**

prototipo: void University::checkDistance(std::string &minor, std::string &major)

Tale metodo si occupa di controllare la distanza presente tra due id successivi passati come parametro e denominati “minor” e “major”. Si salva la dimensione espressa in bit delle due stringhe, “minor” e “major”, rispettivamente in “size\_min” e “size\_major”. Nel caso in cui le due dimensioni siano diverse, ovvero nel caso in cui le due stringhe siano composte da un numero di caratteri e numeri diversi, si lancia una eccezione mostrando la presenza di un errore nel codice del corso.

Nel caso in cui siano uguali si procede con l’analisi della distanza.

DUBBIO Dove ho sottolineato non dovrebbe andare l'indice i. Perché l'indice i non compare nelle righe dopo quindi il for dopo ci la sugli stessi. i dovrebbe andare anche nella seconda condizione del for al posto di size\_maj

* **controlReciprocyGrouped**

prototipo: void University::controlReciprocyGrouped()

Tale metodo ha l’obiettivo di controllare se c’è reciprocità tra i corsi raggruppati: se un corso risulta raggruppato con un altro allora anche quest’ultimo deve esserlo con il primo diversamente viene sollevata una eccezione.

Il metodo si articola nel seguente modo:

* attraverso il primo for, per ogni corso presente nel database, si crea una mappa all’interno della quale si salva per ogni anno accademico un vettore contenente tutti i corsi raggruppati di quell’anno mediante il metodo getGroupedCourseFromAllYear; tale mappa viene denominata appunto “allGroupedCourse”;
* successivamente con il secondo for annidato nel primo si cicla sulla mappa appena creata, “allGroupedCourse”, e in particolare per ogni anno accademico si utilizza un vector al cui interno si salvano i corsi raggruppati aggiungendo in coda il corso considerato detto “groupedOfThisYear”;
* a questo punto per ogni corso raggruppato di “groupedOfThisYear” si procede coll’eseguire lo stesso passaggio effettuato nel ciclo esterno: mediante un terzo for, annidato al secondo, si mette a disposizione un altro vector contenente contenenti i corsi raggruppati a quello puntato dal for in quel determinato ciclo ottenendo così “allGroupedCourse2”; quest’ultimo pertanto contiene tutti i corsi raggruppati aggiungendo in coda, mediante il metodo push, il corso stesso;
* il quarto ed ultimo for viene creato quindi con il semplice scopo di confrontare il contenuto dei due vector realizzati: “allGroupedCourse” “allGroupedCourse2”;

ad ogni ciclo di for si confrontano i due vector e nel momento in cui un corso raggruppato compare in entrambi si procede con l’eliminazione, mediante il metodo erase, dal vector “allGroupedCourse2”;

se un corso raggruppato compare solo in uno dei due e non entrambi come ci si aspetta allora si solleva una eccezione con la quale si rende l’utente al corrente del fatto che la reciprocità dei corsi non è rispettata indicando che ciò accade per il corso.

Se al termine di questo quarto for il vector “allGroupedCourse2” contiene ancora elementi allora vuol dire che la reciprocità non è nuovamente soddisfatta in quanto c’è un corso che presenta dei corsi raggruppati in più;

* **dbCourseNotActive**

prototipo: void University::dbCourseNotActive()

Tale metodo fa in modo di tenere traccia all’interno di un apposito file delle informazioni riguardanti i corsi spenti; informazioni quindi inerenti sessioni precedenti alla data di disattivazione e che, come è stato accennato nei metodi precedenti, sono contenute in “\_tempInfoNotActiveCoursesToWriteInTheDB”; tali info infatti se non scritte in un file ma lasciate salvate in un vector, come “\_tempInfoNotActiveCoursesToWriteInTheDB”, andrebbero perse in fase di terminazione del programma impedendo l’elaborazione di eventuali sessioni di esami relative ad anni in cui un corso, attualmente spento, era attivo .

Si dichiara quindi un file detto “fout” che, mediante il metodo open, apre il file database di nome “offCourses\_db.txt” in entrambe le seguenti modalità:

* out affinché si possa scrivere su tale file;
* trunc affinché eventuali contenuti già presenti nel file vengano sovrascritti;

Mediante l’ausilio di un for si punta ai diversi elementi del vector “\_tempInfoNotActiveCoursesToWriteInTheDB” andando a scandagliare quindi tutte le info relative ai corsi spenti. In particolare tale vector contiene in ogni suo campo le seguenti tre informazioni:

* codice del corso;
* anno accademico a partire dal quale si è spento;
* anno e semestre di appartenenza (ad esempio 1 – 2 nel caso di un corso del 1o anno 2o semestre);

Per ogni ciclo for si procede con la scrittura sul file “fout” dei diversi campi del vector andando a capo con un endl non appena si passa all’elemento successivo. Alla fine di tale metodo pertanto il file “offCourses\_db.txt” dedica ogni sua riga alle tre info del corso spento.

Come buona norma di programmazione il metodo si conclude con la chiusura del file usando l’apposito metodo close.

* **readCourseNotActive**

prototipo: void University::readCourseNotActive()

Questo metodo si occupa della lettura dal file “offCourse\_db.txt” delle informazioni di corsi diventati spenti.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso (addebitata all’assenza del file stesso), grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio detto “line”, viene salvata di volta in volta in coda alle altre info nel vector “\_tempInfoNotActiveCoursesToWriteInTheDB”.

Sempre la stringa “line” viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “token” in modo che ogni elemento del vector contenga l’informazione presente nella riga compresa tra due caratteri separatori “ ; ”:

* token [0] contiene il codice del corso spento;
* token [1] contiene l’anno accademico a partire dal quale è diventato spento;
* token [2] contiene l’anno e il semestre di appartenenza;

Infine per il corso spento letto nella riga si va ad assegnare il semestre a cui apparteneva e l’anno accademico a partire dal quale è diventato spento mediante il metodo assignYY\_Sem.

* **versioning**

prototipo: bool University::versioning(std::string v)

Questo metodo permette di gestire l’operazione di aggiornamento dei database già esistenti con l‘inclusione di informazioni aggiuntive inerenti gli studenti, i professori e le aule.

A tale metodo si passa come parametro una stringa contenente il numero corrispondente alla versione che si vuole mettere in atto e mediante il metodo stoi si trasforma tale stringa in un intero detto “version”:

* se version = 2 allora la procedura di aggiornamento riguarda il database degli studenti e dei professori ai quali vengono aggiunte le seguenti info: data di nascita ed indirizzo di casa per entrambi, data di immatricolazione per gli studenti e data di immissione di ruolo per i professori;

ai prof e agli studenti vengono aggiunti ad entrambi data di nascita ed indirizzo di casa mentre solo

* se version = 3 allora la procedura di aggiornamento riguarda le aule alle quali vengono aggiunte le info riguardanti le strumentazioni necessarie per sostenere un esame;

Si usa poi il metodo renameOldDataBase (spiegato in seguito per chiarezza espositiva) che si occupa del rinominare i database già esistenti aggiungendo “\_old” tra il nome del file e l’estensione “.txt”.

A seconda del valore assunto dalla variabile “version” si procede con lo scrivere i nuovi database contenenti le info aggiuntive. Si ricorda che le info aggiuntive se non modificate dall’utente mediante un apposito metodo vengono salvate (e pertanto anche scritte nel nuovo file database) con dei valori di default. In particolare:

* se version = 2 allora si procede con la scrittura del database degli studenti e dei professori (db\_studenti.txt e db\_professori.txt ) che pertanto adesso conterranno queste nuove info;
* se verison = 3 allora si procede con la scrittura del database delle aule che pertanto adesso conterrà queste nuove info (db\_aule.txt);

Il tutto si conclude con il metodo writeVersion che scrive su un apposito file la versione adottata affinchè qualora il programma dovesse terminare si tiene traccia del contenuto che possedeva la variabile \_version.

* **renameOldDataBase**

prototipo: bool University::renameOldDataBase(int version)

Questo metodo è stato creato per rinominare i database già esistenti aggiungendo “\_old” tra il nome del file e l’estensione “.txt”. A seconda del valore che assume version si decide su quale database rivolgere l’operazione di modifica del nome del file:

* se version = 2 occorre rinominare i database degli studenti e dei professori e siccome i nomi dei file sono noti a priori si dichiarano due vettori di caratteri per database:

“oldname [ ]” = “db\_studenti.txt” “newname [ ]” = “db\_studenti\_old.txt”

“oldname2 [ ]” = “db\_professori.txt” “newname2 [ ]” = “db\_professori\_old.txt”

Si fa uso infine del metodo rename, apposito per i file, che rinomina i file database degli studenti e dei

professori con i nuovi nomi (“newname [ ]” e “newname2 [ ]”) al posto dei precedenti (“oldname [ ]” e

“oldname2 [ ]”).

Nel caso in cui i metodi rename restituiscono dei valori diversi da zero significa che il metodo non è andato a buon fine pertanto si solleva una eccezione con la quale si comunica che i file non sono stati rinominati adeguatamente.

se version = 3 l’operazione di aggiornamento è dedicata al database delle aule e siccome il nome del file è noto a priori si dichiarano due vettori di caratteri:

“oldname [ ]” = “db\_aule.txt” “newname [ ]” = “db\_aule\_old.txt”

Si fa uso infine del metodo rename, apposito per i file, che rinomina il file database delle aule con il nuovo

nome (“newname [ ]”) al posto del precedente (“oldname [ ]”).

Nel caso in cui il metodo rename restituisce un valore diverso da zero significa che il metodo non è andato a buon fine pertanto si solleva una eccezione con la quale si comunica che il file non è stato rinominato adeguatamente.

* **addStudyPlan**

prototipo: bool University::addStudyPlan(std::string fin)

MODIFICARE

* READ con \_version = 3
* ADD con \_veriosne = 2 e 3
* UPDATE con version=2 e 3

INIZIARE LA GESTIONE ESAMI DA: void University::readSessionAcYear()

che sta dopo la dbCOurseWrite()

|  |  |  |  |
| --- | --- | --- | --- |
| STUDENTI | SISTEMA OPERATIVO | COMPILATORE | VERSIONE COMPILATORE |
| Andrea Chiapparo |  |  |  |
| Luca Montangero |  |  |  |
| Ricardo Maria Galiano |  |  |  |