11.5 Case Study: Retrieving Student Information

Once we are able to create classes, we can represent complex objects in software. In this section, the problem is to build an information retrieval system that a university registrar might use to maintain student records.

PROBLEM: INFORMATION RETRIEVAL

The registrar at IO University has a data file named `students.txt` that contains student records:

```
111223333 Bill Board
Freshman    16.0  3.15

666554444 Jose Canusee
Sophomore   16.0  3.25

777889999 Ben Dover
Junior      16.0  2.5

333221111 Stan Dupp
Senior      8.0  3.75

444556666 Ellie Kat
Senior      16.0  3.125

999887777 Isabelle Ringing
Junior      16.0  3.8

      .
      .
      .
```

Each pair of lines in this file has the form

studentNumber firstName lastName
studentYear credits gradePointAverage

where

studentNumber is a 9-digit (integer) student ID number,
firstName, *lastName*, and *studentYear* are character strings,
credits is the (real) number of credits this student carried this semester, and
gradePointAverage is the (real) grade point average of this student this semester.

The registrar at IOU needs a program that will let her enter student numbers, and that will retrieve and display the information for those students.

OBJECT-CENTERED DESIGN

BEHAVIOR. The program should read a sequence of students from the input file `students.txt`. It should then repeatedly prompt for and read a student ID number from the keyboard, search the sequence of students for the position of the student with that student ID number, and if found, display the information for that student.

OBJECTS. An abbreviated list of the objects in this problem is as follows:

	Software Objects		
Problem Objects	**Type**	**Kind**	**Name**
A sequence of students	`vector<Student>`	varying	*studentVec*
Name of the input file	`string`	constant	*INPUT_FILE*
A student ID number	`long`	varying	*studentID*
The position of the student	`vector<T>::iterator`	varying	position
A student	`Student`	varying	none

As we saw in Chapter 10, we can use a `vector<T>` to store a sequence of objects of type `T`. However, we need to store a sequence of students. Since there is no predefined type that allows us to represent a student object, we will need to build a `Student` class for this.

OPERATIONS. The operations needed to solve this problem are as follows:

i. Read a sequence of students from the input file
ii. Display a prompt
iii. Read a long integer from the keyboard
iv. Search a sequence of students for one with a particular ID number
v. Display a student
vi. Repeat steps ii–v an arbitrary number of times

We can accomplish step iv using the Standard Template Library's `find()` algorithm. However `find()` requires that the relational operators < and == be defined for objects being compared, so we add these operations to our list:

vii. Compare two `Student` objects using <
viii. Compare two `Student` objects using ==

ALGORITHM. Given a `Student` class that provides the appropriate operations, we can organize these operations into the following algorithm:

Algorithm for Student Information Retrieval

1. Read a sequence of students from *INPUT_FILE* into *studentVec*.
2. Repeatedly do the following:
   a. Prompt for and read *studentID*.
   b. Search *studentVec* for the student with *studentID*, returning its *position*.
   c. If the search was successful
      Display the student at *position*.
      Otherwise
      Display an error message.

Before we can code this algorithm, we must build a `Student` class. From an internal perspective, the behaviors required of a `Student` include:

- Initialize myself with default values
- Initialize myself with explicitly supplied values
- Read my attributes from an `istream` and store them in me
- Display my attributes using an `ostream`
- Compare myself and another `Student` using the < and == relational operators

These are the minimal operations needed to solve the problem. To make the class truly reuseable, we should add (at least) the following operations:

- Access any of my attributes
- Compare myself and another `Student` using the `!=`, `>`, `<=`, and `>=` operators

The `Student` attributes required to solve this problem include the attributes stored in the input file:

my id number, my first name, my last name, my year, my credits, and my GPA

The following listing of the file *Student.h* shows the class declaration containing prototypes for these operations and instance variables for these attributes. It also contains inlined definitions of the simple operations.

Case Study 11.5-1 The Header File for Class `Student`.

```
/* Student.h declares class Student.
 * ...
 ****************************************************/
```

```
#ifndef STUDENT                              // compile-once
#define STUDENT                              //  wrapper

#include <iostream>                          // istream, ostream
#include <string>                            // string
using namespace std;

class Student
{
 public:                                     // The Interface
                                             //  constructors
   Student();
   Student(long idNumber, const string & firstName,
            const string & lastName, const string & year,
            double credits, double gpa);
                                             //  accessors
   long   getID() const;
   string getFirstName() const;
   string getLastName() const;
   string getYear() const;
   double getCredits() const;
   double getGPA() const;
                                             //  relational ops
   bool operator== (const Student& rightOperand) const;
   bool operator!= (const Student& rightOperand) const;
   bool operator<  (const Student& rightOperand) const;
   bool operator>  (const Student& rightOperand) const;
   bool operator<= (const Student& rightOperand) const;
   bool operator>= (const Student& rightOperand) const;
                                             //  I/O
   void read(istream& in);
   void print(ostream& out) const;

 private:                                    // Implementation Details
                                             // Examples:
  long   myIDNumber;                         //   123456789
  string myFirstName,                        //   Jane
         myLastName,                         //   Doe
         myYear;                             //   Senior
  double myCredits,                          //   15.0
         myGPA;                              //   3.75
};



// ******** Member Operations ***********************************
// -------- Accessors ---------------------------------
inline long Student::getID() const
{
   return myIDNumber;
}
```

```
inline string Student::getFirstName() const
{
   return myFirstName;
}

inline string Student::getLastName() const
{
   return myLastName;
}

inline string Student::getYear() const
{
   return myYear;
}

inline double Student::getCredits() const
{
   return myCredits;
}

inline double Student::getGPA() const
{
   return myGPA;
}


// -------- relational operators --------------------------------
inline bool Student::operator==(const Student & rightOperand) const
{
   return myIDNumber == rightOperand.getID();
}

inline bool Student::operator!=(const Student & rightOperand) const
{
   return myIDNumber != rightOperand.getID();
}

inline bool Student::operator<(const Student & rightOperand) const
{
   return myIDNumber < rightOperand.getID();
}

inline bool Student::operator>(const Student & rightOperand) const
{
   return myIDNumber > rightOperand.getID();
}

inline bool Student::operator<=(const Student & rightOperand) const
{
   return myIDNumber <= rightOperand.getID();
}
```

```
inline bool Student::operator>=(const Student & rightOperand) const
{
   return myIDNumber >= rightOperand.getID();
}

#endif
```

The more complicated operations are defined in the following implementation file *Student.cpp*, for separate compilation.

Case Study 11.5-2 The Implementation File for Class `Student`.

```
/* Student.cpp implements the non-trivial Student operations.
 * ...
 ************************************************************/

#include "Student.h"                        // class Student
#include <iomanip>                          // setw, setprecision
using namespace std;

// -------- default-value constructor --------------------------
Student::Student ()
{
   myIDNumber = 0;
   myFirstName = "";
   myLastName = "";
   myYear = "";
   myCredits = 0.0;
   myGPA = 0.0;
}

// -------- explicit-value constructor -------------------------
Student::Student (long idNumber, const string & firstName,
                  const string & lastName, const string & year,
                  double credits, double GPA)
{
   myIDNumber = idNumber;
   myFirstName = firstName;
   myLastName = lastName;
   myYear = year;
   myCredits = credits;
   myGPA = GPA;
}
```

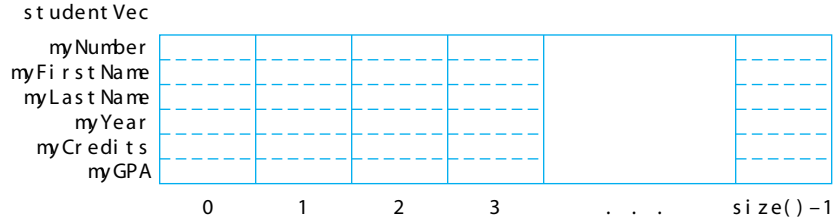
```
// -------- input (function member) ---------------------------
void Student::read(istream & in)
{
   in >> myIDNumber >> myFirstName >> myLastName
      >> myYear >> myCredits >> myGPA;
}

// -------- output (function member) --------------------------

void Student::print(ostream & out) const
{
   out << setw(9) << myIDNumber << '\t'
       << myFirstName << ' ' << myLastName
       << '\n' << myYear
       << setprecision(4) << showpoint
       << fixed << setw(8) << myCredits
       << setw(8) << myGPA << endl;
}
```

Note that the layout of the input file determines the arrangement of the data members in the input statement in function `read()`. In particular, `read()` assumes that the student's ID number comes first, followed by the student's name (first, then last), followed by the remainder of the student's data (year, semester hours, and GPA) on the next line. The output function produces output having a similar format.

CODING. Given class `Student` and the Standard Template Library, our algorithm is relatively easy to implement. The basic idea is to define a `vector<Student>` object named `studentVec` to store the sequence of `Student` values from the input file:

```
vector<Student> studentVec;
```

The effect of this is to create a vector of student objects that we can visualize as follows:



We can then apply any of the `vector<T>` operations described in chapter 9 and the STL algorithms such as `sort()` or `find()` to `studentVec`.

The following program gives the implementation of our algorithm using this approach.

Case Study 11.5-3 Student Information Retrieval.

```
/* registrar.cpp retrieves a student's data from a file
 *  using their id #.
 * Input (file):     a sequence of Students
 * Input (keyboard): one or more student numbers
 * Output:           that student's data
 ************************************************************/

#include <iostream>                         // cin, cout
#include <fstream>                          // ifstream, ofstream
#include <cassert>                          // assert()
#include <vector>                           // vector<T>
#include <algorithm>                        // find
using namespace std;
#include "Student.h"                        // Student


void fill(vector<Student>& sVec, const string& fileName);


int main()
{
  const string INPUT_FILE = "students.txt";

  cout << "This program provides an information retrieval system\n"
          "  by reading a series of student records from "
       << '\'' << INPUT_FILE << '\''
       << "\n  and then allowing retrieval of any student's data.\n";

  vector<Student> studentVec;
  fill(studentVec, INPUT_FILE);

  long studentID;                           // the student we seek
  vector<Student>::iterator position;       // position of the student

  for (;;)                                  // repeat:
  {
     cout << "\nEnter the ID # of a student (eof to quit): ";
     cin >> studentID;                      //  get student's #

     if (cin.eof()) break;                  //  if finished, quit

     position = find(studentVec.begin(),    //  search
                     studentVec.end(),      //   the vector for
                     Student(studentID,     //   this student #
                             "", "",        //   using placeholder
                             "", 0, 0));    //   arguments
```

```
     if (position != studentVec.end())     //   if found
     {
        cout << '\n';
        (*position).print(cout);           //     display student
        cout << endl;
     }
     else                                  //   otherwise, tell user
        cerr << "\nThere is no student with ID # "
             << studentID  << ".\n";
  }
}

// --- Definition of fill() ---
void fill(vector<Student>& sVec, const string& fileName)
{
  ifstream fin( fileName.data() );
  assert( fin.is_open() );
  sVec.reserve(0);

  Student aStudent;
  for (;;)
  {
     aStudent.read(fin);
     if ( fin.eof() ) break;
     sVec.push_back(aStudent);
  }

  fin.close();
}
```

Listing of input file **students.txt:**

```
111223333 Bill Board
Freshman  16.0 3.15

666554444 Jose CanuSee
Sophomore 16.0 3.25

777889999 Ben Dover
Junior    16.0  2.5

333221111 Stan Dupp
Senior    8.0 3.75

444556666 Ellie Kat
Senior    16.0 3.125

999887777 Isabelle Ringing
Junior    16.0  3.8
```

Sample run:

```
This program provides an information retrieval system
  by reading a series of student records from 'students.dat'
  and then allowing retrieval of any student's data.

Enter the ID # of a student (eof to quit): 333221111

 333221111 Stan Dupp
      Senior 16.0000  3.7500

Enter the ID # of a student (eof to quit): 123456789

There is no student with ID # 123456789.

Enter the ID # of a student (eof to quit): 999887777

 999887777 Ringing, Isabelle
      Junior  8.0000  3.8000

Enter the ID # of a student (eof to quit): ^D
```

Note that we use the statement:

```
(*position).print(cout);
```

to display the data of the `Student` being accessed. The parenthesized subexpression:

```
(*position).print(cout);
```

dereferences `position`, the iterator returned by the STL `find()` algorithm, producing the `Student` to which `position` "points" (i.e., the `Student` that `find()` found). The second subexpression:

```
(*position).print(cout);
```

sends the `print()` message to that `Student`, which responds by displaying itself.

Taking the time to implement an object as a class is an *investment for the future*—if the registrar subsequently asks us to write a program to create a list of all students who will be graduating with honors, our `Student` class makes this easy:

```
// ...open inStream and outStream...

cout << "Seniors whose GPA is 3.5 or greater:\n";
for (;;)
{
   aStudent.read(inStream);

   if ( inStream.eof() ) break;
```

```
      if (aStudent.Year() == "Senior" && aStudent.GPA() >= 3.5)
      {
         aStudent.print(outStream);
         outStream << endl;
      }
   }

   // ... close inStream and outStream ...
```

By planning for the future when we design a class, we save ourselves and others a great deal of time and effort.

Note that thus far, the I/O for these classes is a bit clumsy. Code like:

```
cout << '\n';
(*position).print(cout);
cout << endl;
```

is far less elegant than would be the case if the output operator were overloaded:

```
cout << '\n' << *position << endl;
```

See the *OBJECTive Thinking* section that follows to see how to overload the input and output operators.

Exercises 11.5

1. Suppose that a `Student` object *`s1`* will be considered to be less than another `Student` object *`s2`* if:

   the `myLastName` member of *`s1`* is less than the `myLastName` member of *`s2`*; or

   the `myLastName` member of *`s1`* is equal to the `myLastName` member of *`s2`* and the `myFirstName` member of *`s1`* is less than the `myFirstName` member of *`s2`*.

   A rule for the greater-than relationship between two `Student` objects is similar. Provide definitions for `operator<` and `operator>` that implement these relationships.

2. Suppose that a `Student` object *`s1`* can be described as "equal to" another `Student` object *`s2`* if the `myLastName` member of *`s1`* is equal to the `myLastName` member of *`s2`* and the `myFirstName` member of *`s1`* is equal to the `myFirstName` member of *`s2`*. A rule for the inequality relation of two `Student` objects is similar. Provide definitions for `operator==` and `operator!=` that implement these relationships, being sure to clearly state the assumptions they make.

3. An alternative approach to equality of `Student` objects is to describe a `Student` object *`s1`* as "equal to" another `Student` object *`s2`* if the `myIDNumber` member of *`s1`* is equal to the `myIDNumber` member of *`s2`*. A rule for the inequality relation of two `Student` objects is similar. Provide definitions for `operator==` and `operator!=` that implement these relationships. What advantages are there to implementing these operations in this way rather than as described in Exercise 2? What disadvantages? (*Hint:* Consider the problem of searching for a particular student in the registrar's file.)

4. Using the functions defined in Exercises 1 and 2 (or 3), provide definitions for `operator<=` and `operator>=` that implement the less-than-or-equal and greater-than-or-equal relations on two `Student` objects.