CS 215 - Fall 2019
Project 3 – Course Registration

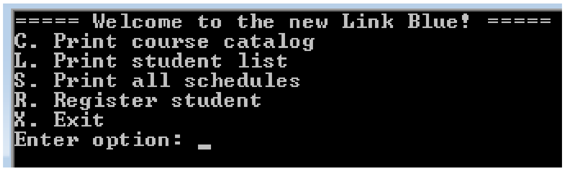
Learning Objectives:

- Object-Oriented implementation of an application
- Text file input and output

General Description:
Implement an application that will allow the user to register students for courses at a university.

The application will start with a list of student records, a list of courses offered (called a catalog), and a list of schedules of students already registered. Data for these three lists will be read from files on startup. Changes will be made to the list of schedules, and the file will be rewritten when the application exits. No changes are made to the course or student lists.

Startup and Main Menu:
The application starts by reading the files then presenting the user with a Main Menu with the options shown in the example here. Of course, the menu repeats the "Enter option: " operation until the user enters a valid option, with an error message for each invalid entry.



```
===== Welcome to the new Link Blue! =====
C. Print course catalog
L. Print student list
S. Print all schedules
R. Register student
X. Exit
Enter option: _
```

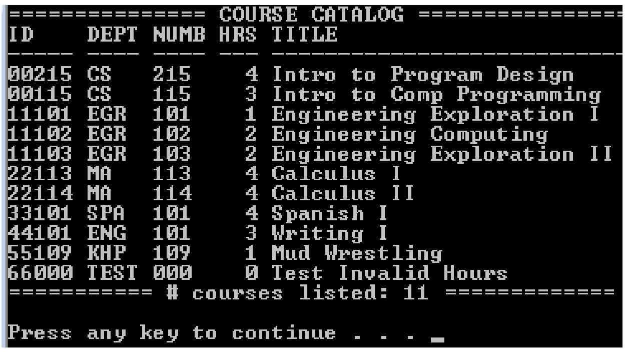
Once an option is selected and performed, the main menu repeats until the user selects **Exit**.

Course Catalog:
Each course in the catalog has the fields (data members):

- **ID** – a string of length 5
- **Department** – a string of max length 4
- **Number** – a string of max length 4
- **Hours** – number of credit hours for the course
- **Title** – a string that may contain spaces

The ID is the "key" to each course, a unique identifier. You may assume the ID, Department and Number do not contain spaces.

The catalog contains a list of a maximum of 50 courses.



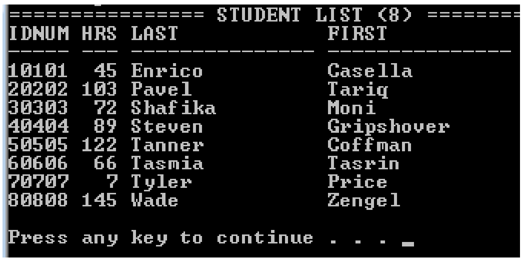
```
=============== COURSE CATALOG ================
ID    DEPT NUMB HRS TITLE
----- ---- ---- --- ---------------------------
00215 CS   215    4 Intro to Program Design
00115 CS   115    3 Intro to Comp Programming
11101 EGR  101    1 Engineering Exploration I
11102 EGR  102    2 Engineering Computing
11103 EGR  103    2 Engineering Exploration II
22113 MA   113    4 Calculus I
22114 MA   114    4 Calculus II
33101 SPA  101    4 Spanish I
44101 ENG  101    3 Writing I
55109 KHP  109    1 Mud Wrestling
66000 TEST 000    0 Test Invalid Hours
=========== # courses listed: 11 ============

Press any key to continue . . . _
```

The print should display the first three "header" lines of the report, and a "footer" line at the bottom with the number of courses in the catalog. Fields should be lined up as shown in the example. A system pause is used at the end of the report.

Student List:
Each student record has the following fields:
- ID – a string of length 5
- Hours – total credit hours earned
- Last Name – a string of max length 15
- First Name – a string of max length 15

The Student ID is the "key" to each student record. You may assume the string fields never contain spaces.



```
=============== STUDENT LIST (8) =======
IDNUM HRS LAST            FIRST
----- --- --------------- ---------------
10101  45 Enrico          Casella
20202 103 Pavel           Tariq
30303  72 Shafika         Moni
40404  89 Steven          Gripshover
50505 122 Tanner          Coffman
60606  66 Tasmia          Tasrin
70707   7 Tyler           Price
80808 145 Wade            Zengel

Press any key to continue . . .
```

The student list contains a maximum of 50 students.

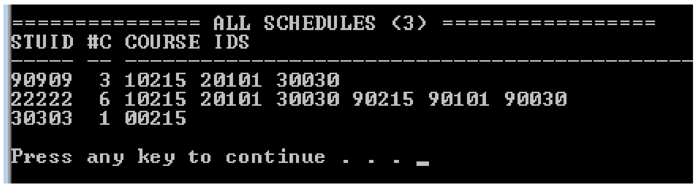
The application prints the student list as shown with data lined up in columns. The first header line contains the total number of students in the list, and the report ends with a system pause.

Schedule List:
A schedule is composed of the following fields:
- Student ID (a string)
- Registered Courses – a list of Course IDs (strings), a max of 10.

Note that the schedule data does not contain details about the student, nor details about the courses. A basic Database Design principle is to store such details *once* on the database (in the student list and catalog), and to not unnecessarily replicate data elsewhere. Relationship files contain only the *keys* of records from other files.

A schedule list contains a maximum of 50 schedules.

The schedule report is primarily a debugging (and grading) function…it is not "pretty". Each line contains the student id and number of courses registered, followed by a list of course id's printed with a space between. The number of schedules (students registered) is shown in the first header line, and the report ends with a system pause.



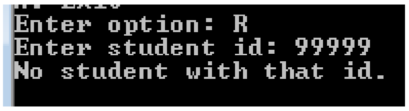
```
=============== ALL SCHEDULES (3) ==================
STUID #C COURSE IDS
----- -- -----------------------------------------------
90909  3 10215 20101 30030
22222  6 10215 20101 30030 90215 90101 90030
30303  1 00215

Press any key to continue . . .
```

In the example shown, there are three schedules with 3, 6 and 1 courses registered respectively.

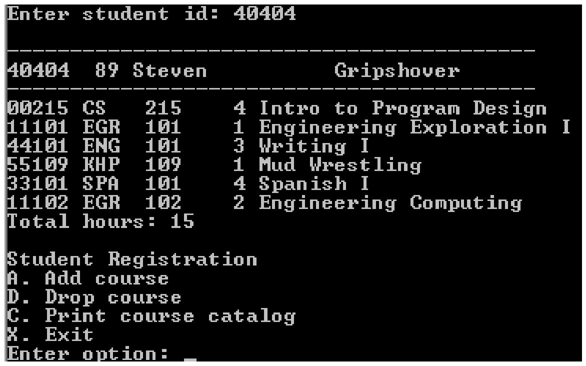
Registering a student:
The objective of this operation is to gather data to add a schedule to the schedule list, **or** update a schedule already in the list.

First, the user is asked to enter a student id. The student list is searched, and if no student with the given id is found, an error message is printed and the operation is completed (the Main Menu will then repeat).



```
Enter option: R
Enter student id: 99999
No student with that id.
```

When the student ID is valid, the schedule list is searched to determine if the student has already registered. If found, this operation changes the schedule already in the schedule list; otherwise, a new schedule is populated and added to the schedule list. The program should print an error message when the maximum number of students registered is exceeded (and no new schedule added, of course).



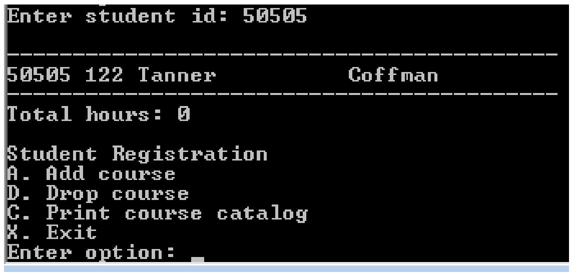
```
Enter student id: 40404
------------------------------------------
40404  89 Steven          Gripshover
------------------------------------------
00215 CS   215    4 Intro to Program Design
11101 EGR  101    1 Engineering Exploration I
44101 ENG  101    3 Writing I
55109 KHP  109    1 Mud Wrestling
33101 SPA  101    4 Spanish I
11102 EGR  102    2 Engineering Computing
Total hours: 15

Student Registration
A. Add course
D. Drop course
C. Print course catalog
X. Exit
Enter option: _
```

The following is then repeated until the user selects Exit:

- the current schedule is printed
- a menu is displayed allowing the user to choose one of 4 options
- the option selected is performed

In the first example shown, the student (40404) has already registered, so the list of courses is displayed. In the second, the student (50505) has not registered yet, so a schedule with no courses is displayed.

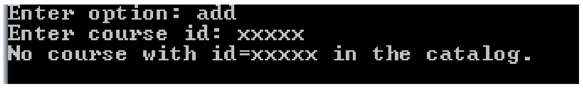


```
Enter student id: 50505
------------------------------------------
50505 122 Tanner          Coffman
------------------------------------------
Total hours: 0

Student Registration
A. Add course
D. Drop course
C. Print course catalog
X. Exit
Enter option: _
```

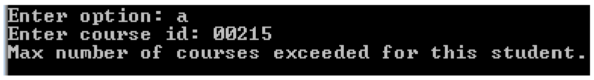
Upon exit, the program should ensure the new/updated schedule has been updated in or added to the schedule list.

Add Course:
The user is asked to enter the course ID. The catalog is searched by course ID, and if not found, an error message is printed and the Student Registration menu repeats.

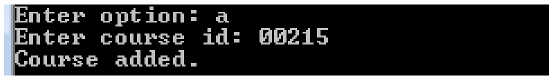


```
Enter option: add
Enter course id: xxxxx
No course with id=xxxxx in the catalog.
```

When the course is found, a new course ID is added to the schedule if there is room. If not, an error messages is printed and the Student Registration menu repeats.

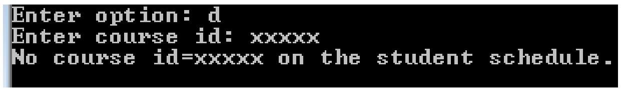


```
Enter option: a
Enter course id: 00215
Max number of courses exceeded for this student.
```



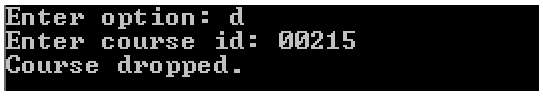
```
Enter option: a
Enter course id: 00215
Course added.
```

Drop Course:
The user is asked enter the course ID. The schedule is searched by Course ID, and if not found, an error message is printed and the Student Registration menu repeats.



```
Enter option: d
Enter course id: xxxxx
No course id=xxxxx on the student schedule.
```

When the course is found in the schedule, the course is deleted from the schedule, a message is printed and the Student Registration menu repeats.



```
Enter option: d
Enter course id: 00215
Course dropped.
```

Print Course Catalog:
This is the same as the main menu's print course catalog. After the catalog is printed, the Student Registration menu repeats.

Object-Oriented Design Specifications:

Objects/Classes in this application consist of:

- a Student	- defines the data and operations for one student record
- a Course	- defines the data and operations for one course record.
- a Schedule	- defines the data and operations for one schedule.
- a Student List	- a partial array of Students and operations on that list.
- a Catalog	- a partial array of Courses and operations on that list.
- a Schedule List	- a partial array of Schedules and operations on that list.
- a Menu	- a menu utility designed to ease the creation/use of menus
- a User Interface	– declares the above objects and interacts with the user to make the application work.

For all classes in general:

A class should be written for each of the above. All classes should include (unless specified otherwise):

- a **header file** (.h) containing the class interface with the name *className*.h
- a **source file** (.cpp) containing the class implementation with the name *className*.cpp
- all data members should be **private**
- **set()** and **get()** methods for each data member (unless specified otherwise)
- **constructor** which sets all data members to some empty value:
    - **string members:** set to the empty string ""
    - **numeric members:** set to 0
    - **partial arrays:** set the number of items in the list to 0; make the MAX a global constant in the class' header file (.h)
- all **read()** and **write()** methods should check if the file failed to open, and if so, prints an error message. After the file is read/written successfully, the file should be closed.

Student class:

- **Data Members**: id, first name, last name and total credit hours (see details above)
- standard constructor and set()/get() methods
- **print() method –** prints the data members on one line, with a space between, using the following field widths and justifications:

data member	print width	justification
id	5	left
credit hours	3	right
last name	15	left
first name	15	left

Course class:

- **Data Members:** id, department, number, credit hours, title (see details above).
- standard constructor and set()/get() methods, except:
    - setCreditHours(int hrs): if the number of hours is less than 0, set to 0 instead.
- **print() method –** prints the data members on one line, with a space between, using the following field widths and justifications:

data member	print width	justification
id	5	left
department	4	left
number	4	left
credit hours	3	right
title	20	left

Schedule class:

- Data Members:
  - student id
  - array of course id's string
  - number of courses (for which this student is currently registered)
- Constructors/sets/gets:
  - standard constructor
  - standard gets() for student id and number of courses
  - getCourseID(int index) – given an index into the array, returns the course id from the array when the index is in bounds; returns empty string when the index is out of bounds.
  - standard set() for student id
- Methods:
  - **add course**: given a course id (assumed to be valid), appends the id to the course array, increments the number of courses, and returns 0 when there is room; returns -1 to indicate the course array is full for this schedule (and so no course id is added).

  - **drop course:** given a course id, searches the course id list. If not found, returns -1. If found, removes the course id from the course id array and returns 0.

  - **print:** prints the data members on one line, with a space between, using the following field widths and justifications:

data member	print width	justification
student id	5	left
number of courses	2	right
each course id	n/a	n/a

Catalog class:

- Data Members:
  - *partial array* of Course objects
  - (which includes a) number of courses

- Constructor/sets/gets: because a data member for this class is an array, the sets/gets are not all "regular". So here is the list:
  - constructor – standard
  - getNumCourses – standard, but there should be no setNumCourses, as the number of courses would be incremented by an "add course" (which we don't need).
  - getCourse – given an index into the array, returns a *Course* object which is a copy of the indicated array element. When the index is out of range, returns a copy of an "empty course":
    ```
    course emptyCourse; //constructor course ID to ""
    return emptyCourse;
    ```
    So this method just gets *one* of the courses; we won't do a method to get ALL (the whole array of) courses.

- Methods:
  - **search**: Given a course ID number (string). Returns the index in the course array where there course matching the given course ID was found, OR returns -1 when not found. (See **userint.cpp** line# 69)

- **print**: Prints the course catalog as shown on page 1. It should first print the title and column headers. It should *then invoke **print()** in each of the course objects in the course array* (do not re-write the `cout` to print the course information on each line…that's what the course::print() does). It should then print the number of courses in the "footer" as shown, followed by a system pause.

- **read:** Opens a text file called "catalog.txt" and reads the course data from the file into the partial array of courses. Note you will have to read data from the file into local variables, then use the course set() methods to add them to the course object. Each line of the file contains data on one course in the format:

```
courseID  department  courseNum  numCreditHours title
```
(the `title` may have spaces).

The last line is an "end of file" (sentinel) record that contains:
```
      XXXXX XXX XXX 0 End of List
```

Sample File: (one is also available on the course website)
```
00215 CS    215   4 Intro to Programming        ID# Dept Num Hours and Title for one cousre (geline() for Title)
00115 CS    115   3 Intro to Comp Prog
11101 EGR   101   1 Egr Exploration I
XXXXX XXX   XXX   0 End of List                 Sentinel Record (End of Data)…so there are only 3 courses, not 4
```

Tip for the read()s:
You should read data for one course (or student or schedule…below) into local variables. Then use the *set()* methods for the classes to store the data in the individual objects of the array.
Example:
```
       f >> id >> dept >> num >> hours;
       getline(f, title);
       courses[i].setId(id);
       courses[i].setDept(dept);
       courses[i].setNumber(num);
       courses[i].setHours(hours);
       courses[i].setTitle(title);
```

This code is just an *example*…copy/paste of the above code probably will not work…slight rework is required.

Student List Class

- Data Members:
  - *partial array* of Student objects
- Constructor/gets/sets
  - standard constructor; no gets/sets will be used, so they may be left out
- Methods:
  - **print:** Prints the Student List as shown in the example above. On the first line, the report title with the Number of Students in parenthesis, followed by the column headers.
    The lines for students should be printed by the student::print() method; do not code a `cout` for those lines. The list is followed by a system pause.

- **search**: given a Student ID, searches the array for a student record with the given ID. When found, returns a copy of the student record from the array. When not found, returns an “empty student” record, which has empty string as the student ID.

- **search:** given an index into the array, returns a copy of the array element with the given index. When the index is out of range, returns an “empty student” record. (Yes, there are two **search()** methods, but they differ by argument list).

  Note: yes, there are two search() methods, which is allowed, as long as they differ by *argument list*. The first is given string (student ID) and the second is given an array index (int).

- **read:** Opens a file called “stuList.txt” and reads student data from this file. The file format is:
  - first line contains the number of students
  - each subsequent line contains data for one student in the format:
    `id    last    first    credits`
  - you may assume there are no spaces in the last and first names.

  Example Input File

  ```
  3
  10101 Enrico Casella 45
  20202 Pavel Tariq 103
  30303 Shafika Moni 72
  ```

  Thee are 3 students at this university
  First student: ID=10101, First=Entrico, Last=Casella, totalCredits = 45

Schedule List Class:

- Data Members:
  - a *partial array* of Schedule objects

- Constructor/sets/gets
  - constructor – standard
  - standard get for number of schedules, but no setter for number of schedules.

  - **getSchedule**: given an index into the array, return a copy of the schedule; return an “empty schedule” object when the index is out of range.

  - **getSchedule**: given a Student Id, searches the array for a schedule with the given student ID. It returns a copy of the schedule found, or an “empty schedule” object when not found. (Yes, there are two getSchedule() methods, but they differ by arguments)
- Methods
  - **addSchedule:** given a schedule object, it first searches the schedule list by student ID to see if the student already has a schedule. If found, it copies the given schedule object into the found schedule element of the array.
    Hint: schedules[foundNdx] = givenSched
    If not found, copies the object’s data into the next available element of the partial array and increments the number of schedules, when there is room. When the schedule list is full, prints an error message.

- **print:** Prints the Schedule List as shown above. The number of schedules appears on the first title line. *The data for each schedule should be printed by schedule::print;* do not code another `cout` to print these lines. The report is followed by a system pause.

- **read:** Reads data from a file called "schedList.txt". The file format is:
  - on the first line: the number of schedules in the file
  - for each schedule, there is one line of data with
    - student ID
    - number of courses for which the student is registered (may be 0)
    - the course IDs of the courses that are registered (0 or more course IDs)

  Read() Example input file:

```
4                                4 schedules so far
90909 3 00215 11101 33101  student 90909 registered for 3 courses: 00215, 11101 and 33101
40404 6 00215 11101 44101 55109 33101 11102  registered for 6 courses
30303 0                               student 30303 registered for 0 courses so far
10101 2 00215 55109                   student 10101 registered for 2 courses so far
```

- **write:** writes the current data to a file called "schedList_1.txt". The file format is identical to the format of the input file "schedList.txt"

Menu class:
Most all menus in any of our projects all work in basically the same way:
- a title of some sort may be displayed
- a list of options is presented
- a validation loop is used to force the user to enter a valid menu option
- the result is a valid menu option that is then passed to other parts of the program.

The purpose of this menu class is to create a **reusable** (in many projects, and in two places in this project) menu class that is "configured" then "invoked" by a program that needs this standard type of menu.
- Data Members:
  - a *partial* array (MAX 10) of valid menu options (characters).
  - a *parallel partial* array of descriptions for each menu option
    Since these two arrays are parallel, only one MAX and one NUM are needed for both.
  - a title, to be displayed at the top of the menu
  - an error message, used when an invalid option is entered by the user

- Constructor: standard initialization of data members

- Sets() and Gets(): to shorten the project, we'll leave most of these out
  - set methods for the title and error message members

- Methods:
  - addOption()
    Given: an option char and an option description
    Returns: nothing
    Adds (appends) the given values to the option and description partial arrays, when there is enough room. Prints a “no room for more options” error message when there is not enough room to add another option.

  - doMenu()
    Given: nothing…it is assumed the menu has been configured using the sets() and addOption().
    Returns: the valid option selected (a char)
    Prints the title string on one line, followed by the options list (option char, a dash, and option description, for each).
    It then asks the user to enter an option, and uses a validation loop to force the user to enter a valid option. The error message member is printed when an invalid option is entered. When a valid option is entered, it is returned by the method.

  - validOption()
    Given: a character
    Returns: true when the given character is found in the options array, or false when it is not found. This method should be invoked by doMenu().

    Example Usage: once completed, an object of the menu class would be used like this:

```
// create and configure the menu
menu m;
m.setTitle("==== What to do  =====";)
m.addOption('S', "Study");
m.addOption('R', "Relax");
m.addOption('P', "Party");
m.setErrorMsg("Dummy! Enter S, R or P!! Try again!");

// later, when ready, just "do" the menu to get the option
char opt = m.doMenu();
if (opt == 'S')
       etc.
```

User Interface class:
Other than printing of reports and a few error messages (and the menu class) there is no interaction with the user in the methods of the above course, student and schedule classes. This “user interface” class uses objects of the above classes and implements the “user interface”…interaction with the user for this program. Note that any of the above classes would be reusable in other programs. This class will implement *this* program/project.

My solution to this class has been provided for you on the course website: userint.h and userint.cpp
If you have already written any of the above classes, note you may have to change the spelling of your classes and public methods to fit the spellings in the code given **or** you may change the spellings in the code given to match the spellings in the code you have already written. The simple **main.cpp** is provided as well.

Submission:

Zip all .h and .cpp files into a .zip file. Do *not* include .txt files (we'll use different ones to test your code) and most especially do not include any of the other MS VS files in the project.

Submit the .zip file in Canvas.