CSCE 315 **Team Project 1**
Fall 2017

Project Goal:
This project has 3 main parts. First, you are to write a set of test cases that can be used to help test implementation of a particular database API. Second, you are to create a standalone "library" that will implement that database API, and that will meet the functionality described in the test cases. Finally, you will implement a basic database application program that makes use of a database implementation provided by a different team.

Note that in a "real" application, it is unlikely that you would implement one of these database systems "from scratch." More likely, you would use an existing database system, and access the system via query functions, a DDL, and/or a DML. This project is meant to help you:

- Understand how a database system might work at a low-level
- Become familiar with the basic database operations and queries, and how these can be used to create an application using database operations
- Understand good data structure choices and algorithmic implementation of basic data structures
- Become familiar with writing tests and regularly testing code
- Become familiar with use of a version control system
- Gain practice in basic modular organization and API definitions, and library specification

Development Specifics:
Your project will consist of three major pieces of code:

1. A set of test scenarios,
2. A database library,
3. And an application module.

A key part of this project will be that you will not develop both the application and the database implementation yourself. You will develop a database implementation for use by another team, and write an application that uses another team's database implementation. Teams will "deliver" their library to a different team. So, each team will be acting as both a producer of their own library and a consumer of another team's.

For this project, the default is that all development should be done in Visual Studio with C++. If you reach an agreement with the other team (whose database system you will be using, or to whom you are providing the database) to instead develop for Linux, you may use Linux instead. The language must still be C++, and other platforms are not allowed.

The details of the code are described below.

Team Organization:
Teams should consist of 3 people, with 1 or 2 exceptions of 4-person teams per section. The specific list of team members for each team, along with the team number, will be posted on the class piazza page, after the students have chosen their teams.

Specification of Database Library:
You should create an independent library that gets compiled into a .lib in windows. Other OS's have similar library packaging features. This library should allow a user to specify a relational database, add data to the database, and perform queries on it. What is given below is a ROUGH API. You will be asked to put together a more complete API specification, including error handling, based on this. The idea for specifying the below is to make it easy for a team to change to a different implementation if needed, but it is not expected that all implementations will be interchangeable.

Your Database library should require the inclusion of a single header file only (along with linking to the appropriate library). Other header files could be distributed along with it, though.

You need to define 3 classes: *Database*, *Table*, and *Record* (you may pick different names); you may include other classes, as well (e.g. you might find an attribute class helpful). All the routines below can use different names, and there will be variations in the way that a routine can be implemented and in the exact interface. You may choose whether to pass references or actual objects as parameters, for instance. But, the functionality should be provided as specified. Note that not all commands are necessarily specified. For example, destructor functions are needed, functions to allow a save might require a save routine for tables, or a query might be allowed as a member function of a table in addition to the database.

Note that I am using the terms "function," "routine," etc. very loosely here and should not be interpreted strictly.

The items marked with a * are required only of the 4-person teams; 3-person teams are not required to implement these.

Database:
Generally, a user should instantiate a single database object and work from there. The database object should include commands to do the following:

- You should allow a constructor function with no arguments (other constructors may be included if desired). This creates an "empty" database.
- An add table function that takes a single table object and a name, and adds that table to the database.
- A drop function that takes a table name and deletes it from the database
- * A save function that takes a file name and saves the database to that file (save format is entirely up to you).
- * A load function that takes in a file name and loads in a database from that file. Any existing database should be deleted.
- * A merge function that allows another database to be merged into that one.
- * A copy function that copies an entire database.
- A list table function that returns a list of all table names in the database
- A get tables function that returns all the tables in the database.
- A query function (described below, separately)

- * A delete command. The structure of the delete command should follow that of the query command, except that instead of returning a table, the table in the FROM portion has the appropriate tuples deleted.
- * An update/modify command. The modify command should take a table name, a WHERE clause (as in the query command), and a SET clause. The SET clause should be able to reference attributes of the table, and set them to values that are either a constant or (in the case integers and floats) a computed function on attribute values (from that table). The operations +, - , *, and / should be supported.

Note that there are multiple ways to handle a returned list of names or the tables. For instance, you might use an array, or a vector, or you might return a set or some other container.
Some example code *might* be (note: this is just one example, and will depend on how you define the functions!):

```
Database db();  // Creates an empty database
Table t(); // Creates an empty table
db.AddTable("Tickets",t);
        // Adds table t to the database, naming it "Tickets"
vector<string> alltables = db.GetListOfTables();
        // Returns just "Tickets" in this example
Table s = db.Query("*", "Tickets", "True");
        // Runs a query on the database, returning a table; see below
```

Table:
Note: Tables only have a name in the database; an individual table will not be named, in general. A table can exist outside the database, but will not have a name or be saved as part of the database save command.

The table class should support the following:

- Constructor of a table. An empty constructor should be allowed, to create a table with no rows or columns.
- A constructor should also be supported that takes a list of attribute names, or use a new structure with a name/type, passed in as arrays, or as some other container, etc.
- An add function that takes in a single attribute name, and adds a column to the *end* of the table with that new attribute. Any entries currently in the table should get NULL for that entry.
- A delete function that takes an attribute name and deletes it from the table.
- An insert command that takes a record and adds it to the table.
- A get attributes command that returns a list of the attributes for that table, in order
- A get size command that returns the number of records in the table
- An iterator of some sort that can be used to return individual records from the table. There are many ways this can be done.
- A function that allows a single attribute name to be designated as a key for the table.
- * A rename attribute command that takes two names, and replaces the name for the attribute given by the first name with the second name.
- * A command to specify a key. The command should take a set of attribute names that will form the key.

- A cross join command that takes two tables as input and produces one table as output.
- A natural join command as follows: Two tables are taken as input, and one is produced as output. The first table should have an attribute name(s) matching the key from the second table. The join should create one entry for each row of the first table, with the additional columns from the matching key in the second table. If the second table does not have a key, or the first table does not have an attribute matching the key name, then an exception can be thrown, or an error returned.
- Routines that take a single attribute name as input, and compute the following:
  - Count (counts non-null entries only)
  - Min
  - Max

Note that there are many ways that some of these commands can be implemented, but the functionality needs to be provided. For example, a table does NOT need to keep records (but might choose to) – it could use a different storage format, but it needs to be able to take in a record and return a record.

For this application, you can assume that all data types are strings (or character arrays, if you choose).

No table should have two attributes with the same name.

Some example code *might* be as follows:

```
Table t("ID", "Name", "Birthdate");
        // Creates a table with 3 attributes
t.AddAttribute("Address");
        // Adds a fourth attribute
t.DeleteAttribute("Birthdate");
        // Removes an attribute
t.SpecifyKey("ID");  // Sets "ID" as being a key for the table
Record r(3);  // Creates a record, empty in this case, of size 3
t.Add(r);  // Adds record r to the table.
```

Record:

You should have a separate record class that is used to store an individual record (tuple). A record should consist of an ordered set of strings. Note that the record should be strings, though those strings might represent an integer, floating point number, date, or time. That is, a record on its own has no "knowledge" of what its entries mean, but sees them all as strings. Note that strings can be implemented as character arrays or using the string class. Also, how the record itself is implemented is up to you (e.g. it could be an array, it could be a container). The routines that need to be provided for records include:

- A constructor that allows creation of a record of arbitrary size, and initialization of the entries to a null string.
- A function to return the size of the record (i.e. how many entries it has).
- An accessor function that allows access to individual entries in the record. This could be the [] operator, for instance. You should allow individual entries to be read, and to be written, and could use one approach for this, or could provide separate read/write routines.

Example code *might* be:

```
Record r(3);
cout << r.size() << endl; // outputs "3"
// The following are all different options, depending on how you
specify accessors
r[2] = "John"
r.set(2,"John")
r.set("One", "Two", "Three");
string s = r[2];
string s = r.get(2);
```

Query:

The query command should be a member function from the database. It will return a table. The query command should take in three arguments, a SELECT argument, a FROM argument, and a WHERE argument. Each argument should be specified as a string to be processed. These three arguments should have the following properties:

- In the SELECT argument, allow either
  - A list of which attributes names to keep. These should be a list, in order, of the
  - An * to keep all attributes.
- In the FROM argument
  - A single table name
- In the WHERE argument, references to the attribute names.
  - Comparisons of =, <>, >, <, >=, <= (string comparisons)
  - * An IN operator (given the name of a table with only one attribute)
  - * An EXISTS operator (given the name of a table with only one attribute)
  - AND, OR, and NOT
  - Parentheses. Parentheses can be nested.
  - * ALL (given the name of a table with only one attribute)
  - * ANY (given the name of a table with only one attribute)

Note that the WHERE clause (and to a lesser degree the SELECT clause) will require parsing.

Checkpoint 1:

a. Specifying the API :

You are to develop an API with your team that describes *precisely* how you will provide the above functionality. Although most of the operations should be clearly described above, there is still flexibility as to exactly how the functionality is provided. Your team should put together a detailed API specification, which will be passed on to the team that will use your database implementation. This should be contained in a single written document (plain text, HTML, PDF, Word, etc.). The API should specify exactly what the arguments to each routine are, what is returned, and identify any error codes or requirements on the input.

With this determined, you should provide a header file and a .lib that implements the API, ***but with no actual functionality***. That is, you should provide a header file and a .lib file that will allow someone to compile (and run) a program that makes all the various calls to your program. All that needs to be supported are the calls – the results of those calls can be total and complete

nonsense. At this point, do not include any real database functionality in your library. This file, though, will allow the other team to write their test cases.
The API document, header file, and .lib need to be passed to the team that will use your library, in addition to being turned in.

b. Team Organization Plan:

You should also provide a short (1-paragraph minimum to 1-page maximum) summary of how your team will be organized. You should clearly state what role each person on your team will have, and who will be responsible for what throughout the project. This needs to be turned in with the .zip file.

Checkpoint 2: Test Creation:

The team that is *receiving* the database system will develop a set of tests that the database developers should run their code against. Essentially, the team receiving the database should write a series of programs (and possibly "dummy" data files) that can be used to test the functionality of the system provided.

At the second checkpoint, the team receiving the database should deliver the code (source code) to the team providing the database, for that team to use in ensuring that their code works.

The team receiving the database should develop (multiple) simple test cases. Each should consist of a program that is run, and asserts several actions as being true. For instance, a program might create a database, create a table, add the table to the database, then return a list of tables, all using commands from the database. Then, the program would compare the table list that came out to ensure it contained just the single table that had been inserted. This would be a very basic test for adding one table; other tests will be needed to test adding multiple tables, to test table manipulation and record manipulation, and to test queries.

*** The test programs in this case should either provide an error statement saying that the test failed, or a single statement output to standard output stating "Passed". *** Tests should not test aspects that are not required of the database system.

If the team developing the database is able to achieve "Passed" on all the tests provided, it is assumed that the team receiving the database can make use of the database system provided. If the team providing the database cannot pass all tests provided, and the tests were testing what should have been valid functionality, then the receiving team may have the option of using a different database implementation.

Checkpoint 3: Database Implementation:

At Checkpoint 3, you should provide a complete working library .lib file that implements your database system. This should, at minimum, pass the tests that were given to you by the receiving team at Checkpoint 2. A single document should be produced, stating that the code passes all tests provided (if that is the case). If any of the provided tests were not passed, this should be explicitly stated, in a separate document.

Checkpoint 4: Specification of Application:
Your team will be required to build an application program that uses the database defined by the first team. More details of this will be given later, after the first checkpoint. Allow plenty of time to work on this. But, at the beginning of the project, you should be focused on designing general database functionality, and not something specific to the application.

Team Organization:
For this assignment, you should form your teams as you wish. All team members must be from the same lab section. There should be 3 members per team. If the lab does not divide evenly into groups of 3, then up to 2 teams of four may be created instead. These teams of four will have more requirements for the project itself.

For this project, you may choose, organize and run your team as you would like. In later projects, the team structure and operation will be given more specifically, but in this one, you may organize as you see fit. However, you will be asked to a) describe how you organized your team, b) give a justification for why you decided to organize in that way, and c) provide feedback on how well that organization worked. In other words, you should consciously decide on how you will work together, divide up responsibilities, etc., and not just start working and see how things fall out.

Code Organization and github:
This project is to be completed in C++. The default assumption is that you will develop it using Visual Studio. If, and only if, a team that you are working with (to either receive a library from or that you are providing a library to) agrees, you may develop your code in Linux, instead. This should be resolved by the first checkpoint.

Your team is required to maintain its development using a github repository (at github.tamu.edu) in which your team will keep current versions of source code, documentation, etc. You should **not** pass around files by email, saving in shared directories, etc. Significant points will be taken off of the project if code is shared via a method other than github.

You are to create a Development Log inside the github repository for your project, that is a wiki-like log of the work you have performed on the project. The log should be updated each time there is a significant amount of development added in your work, and entries should include a date (and possibly time) for the entry, the name of the individual making the change, and the work done.

Your team should give access to the github repository to the TA/instructor. The instructor/TA may download the software and monitor progress throughout the project.

Intermediate Stages:
For each of these checkpoints, the key files that are due should be submitted via ecampus, as a snapshot of the current state. For checkpoints 1-3, the files should also be passed to the other team, as appropriate. When submitting code in ecampus, please submit source code and documentation only (i.e. not libraries or executables) in a single zip file.

Checkpoint 1 – Header file and stub code: At this point, you are to deliver to the second team an API description and a header file with dummy library files that they can use to write their test cases. You should have determined if the default Visual Studio will be used, or if a Linux implementation will be allowed instead, and the library files should either be for Linux or Windows, depending on that choice. Also, a document describing team organization should be submitted.

Checkpoint 2 – Test Cases: The receiving team should put together a set of test files/programs that will be used to test the database system. This should be delivered to the database developers, and the files turned in on ecampus.

Checkpoint 3 – Database Delivery: At this checkpoint, the database development team should have developed a "working" version of the database system that passes the test cases given by the receiving team. This should be delivered to the receiving team by sending the header file(s) and the appropriate library files. A document either stating that all tests were passed, or noting those that did not should be provided. Note: you want to work on this from the point Checkpoint 1 is finished, not after Checkpoint 2!

Checkpoint 4 – Application Development: The final checkpoint (end of the project) will require that the working application program be submitted.

Bug reports: If, after checkpoint 3, additional bugs are found, the receiving team should communicate these bugs to the sending team as soon as possible, documenting any requests. Any bug report must include additional test cases that will test a fix for that bug. The database development team should make an effort to deal with these bugs, but this should be headed off by designing thorough test cases to begin with. It is the receiving team's responsibility to create a good set of test code.

Problems with team providing database: If your team encounters serious problems with the database provided by another team – i.e. tests that were valid but did not pass, then you might be switched to a different group as a database provider. If this happens, you will need to communicate with the TA to determine whether this will be allowed, and to ensure you have made every effort to use the other team's library. The TA will identify a different database system that had a thorough set of test cases provided, and that passed all test cases, that you can use instead.

Documentation of Final Project:
Your final report should include two pieces of documentation. In addition, you should submit your code in its final state via ecampus. This submission will be your final graded project, and the turnin time will be based on that code turnin.

First, you should include a document describing the use of your application program (this is the part that will be graded, below). This document should (without being too wordy) describe the features of your application program, including how to perform all of the basic commands desired. Note that your team may be asked to demonstrate your application program shortly after the time it is turned in.

Second, your team should turn in a (1-2 page) summary of the library given to you by the first team. You should clearly specify whether the team met its responsibilities in terms of delivering required pieces (API, code at checkpoints) on time and with the functionality needed. List any difficulties encountered, including any bug reports/fixes that were needed along the way. More information about this will be provided as we near the end of the project.

Turning in Work:
Except as specified otherwise (e.g. emails), you are to submit work and reports via ecampus turnin system. Individual reports (e.g. the final report regarding teamwork) should be submitted by individuals. The team reports and code should be submitted by *one* team member.

Final Report(s):
Teams and individuals will be required to put together a specific report detailing how the teamwork went. These reports will be used to determine the grade allocation among the team members. Teams may either agree on a grade distribution among themselves, or may opt to allow the instructor to determine the allocation. Details of these final reports will be given later.

Project Grading:
The overall project will be graded as described below. This grade will be used to determine the team grade, and individual grades will be determined by apportioning the team grade among the team members (the exact breakdown will depend on the team and individual project reports). Note that your grade on the database library will be determined in part by evaluation by the second team that uses your code.

7% Checkpoint 1 – Is an API fully described? Is the header file and dummy library sufficient for making the API calls?
15% Checkpoint 2 – Are a complete and thorough set of tests provided?
30% Checkpoint 3 – Functionality/completeness/correct operation of database library
10% Evaluation of database library by other team.
25% Checkpoint 4 – Functionality/extent/correct operation of application program
3% Completeness of required reports
10% Code style: naming/layout/commenting

Dates and Deadlines:
The following are the intermediate deadlines for this project. Details of the particular requirements are described above. Note that these are final deadlines; it is likely that your team might want to complete several of these well in advance of the "due date", and you will want to begin work on later checkpoints before the earlier ones. To be clear, **you should not wait until the end of the prior checkpoint before beginning work on later portions of the code!** For instance, it is unlikely that you will be able to complete the work for checkpoint 3 if you do not begin working until checkpoint 2 has passed.

Monday, September 18
Project Assigned

Sunday, September 24, 11:59 p.m.
Checkpoint 1 – Specification of header file(s) and stub library due

Sunday, October 1, 11:59 p.m.
Checkpoint 2 – Full set of testing code due

Sunday, October 15, 11:59 p.m.
Checkpoint 3 – Final database implementation due

Sunday, October 22, 11:59 p.m.
Checkpoint 4 – Final application program due

One day after final code turned in (or **Monday, October 23**, 11:59 p.m.)
Final group report turned in

One day after group report turned in (or **Tuesday, October 24**, 11:59 p.m.)
Individual Reports on teamwork turned in