University of Cape Town
Department of Computer Science

CSC3022H/CSC3023F – C++ Tutorial 1 A Simple Student Database

February, 2015

The objective of this tutorial is to build a simple database of student records, which can be queried for student information. The database will consist of fixed size records which contain student details. You will first need to set up the source code framework/structure and build a simple menu-based interface. After you've successfully done this you can fill in the details by implementing all database functionality.

The database will not be sorted or indexed since this requires more complex algorithms and data structures. This is an exploratory tutorial, intended to give you some experience with basic C++ coding.

You have **one week** to complete this introductory tutorial. **Remember to use local version control from the get go and commit your work regularly.**

Architecture and interface

Build a simple text-based interface which can be used to perform operations on your database. You must write "stubs" so when an option is invoked, the function prints a message like "function QueryDatabase() called". You will provide the correct functions during the second part of the tutorial.

Your database interface must have options to do the following:

1. add student
2. read database
3. save database
4. display given student data
5. grade student

Do not worry about the underlying functions right now; just ensure that when an option is selected, the right "stub" message is printed and the screen is sensibly redrawn. The menu could look something like the following:

```
0:  Add student
1:  Read database
...
q:  Quit
```

`Enter a number (or q to quit) and press return...`

To clear the terminal window, you can use a shell command:

```
void clear(void) { system("clear"); } // include cstdlib
```

Since you are reading menu options continuously, you will need an "event loop" to process your menu selection. You can use a `for` statement to achieve this:

```
for (;;) { // loop forever
...  // process key press and call relevant functions
if (terminate_condition) break;
}
```

Recall that you can use a struct to bundle together multiple fields into a single datatype. The **StudentRecord** `struct` must have the following fields:

Name (String), Surname (String), StudentNumber (String), ClassRecord (String).

The **ClassRecord** field is a string of space separated numbers which reflect various marks the student has obtained during the year. For example: "54 66 72 34"

You must set up the basic code structures you need: a **StudentRecord** struct and a **std::vector** to store a number of **StudentRecord** instances. The **std::vector** data structure is an expanding array-based structure that comes bundled with C++ and you can consult the documentation at `http://www.cplusplus.com/reference/vector/vector/` on how to use it. The vector data structure supports random addressing using the familiar `[]` notation. For readability please ensure that your methods and the vector of records is defined in a separate C++ file with its own header and **NOT** in your driver.

Please ensure that you always use your student id as **namespace** when defining methods, structs and classes in this course:

```
/**
*.h file:
*/
#ifndef DATABASE_H
#define DATABASE_H
//any includes here
namespace STUDENT_NO {
void add_student(std::string name ...);
  ...
}
#endif
/**
*.cpp file:
*/
#include "database.h"
STUDENT_NO::add_student(std::string name ...){
  ...
}
```

Remember that you usually have a basic driver file, containing `main()` and other necessary functionality (such as the event loop) and a collection of class source files, along with appropriate header files for other cpp files.

You must also write a Makefile (or mod the one from the consolidation session) to compile your work. This should be easily extended, since you may need to change it for the next part of the tutorial.

The Database Engine

In this part of the assignment you will provide definitions for all the method stubs you created in Part One.

The functionality required is as follows:

Add student Enter new student data

Read/Save database Write or read a simple text file which stores a list of database entries (see below);

Display student data Print the record for a given student, based on the student number entered;

Grade student Print an average for a student based on a given student number. The average will be constructed by extracting all the number in the ClassRecord string and averaging them.

Exit Exit from the application; any information not saved will be lost.

Handin Date: 3rd March 2015 at 10AM.

Notes:

1. when comparing strings, note that string comparisons are case sensitive (so *Durban* is not the same as *durban*);

2. You must use a `std::vector<>` container to hold your record data; this will require you to include the header file **vector**. The `vector` data is not sorted by default. It is not expected that you will sort the data — a simple sequential search will be adequate for this work;

3. File I/O requires the use of `ifstream` and `ofstream` objects. Basic I/O uses `<<` and `>>` in the same manner as console I/O. For example,

   ```
   #include <fstream>
   ...
   int myint;
   ifstream ifs("inputfile.txt"); // argument is ‘‘char*’’ NOT String
   ifs >> myint;
   ifs.close();
   ```

   opens an input file stream and reads an integer from it, placing it in `myint`. The stream is then closed. Output file streams work in a similar manner. More on this can be found in the notes;

   Do **not** use C-like mechanisms to accomplish this.

4. You can read input from a string (instead of a file): use an *input stringstream:*

```
#include <sstream>
...
string X = "buenos dias mi amigo", value;
istringstream iss(X);
while (!iss.eof())
{
iss >> value;
cout << "value =" << value << endl;
}
```

5. You can write the record data out in the order in which it is stored in the vector. To reconstruct the list, simply read each record in turn and "push" it back onto an empty vector. You must decide on a format for your database file;

6. You may **not** have duplicate records (i.e. two or more records for the same student number). When you enter new data, you must first check to see whether that student exists. If so, overwrite the old data with the new, otherwise create a new record as expected.

7. You must hand in the Makefile and your source file(s). The Makefile must function correctly.

Please Note:

1. A working *makefile* must be submitted. If the tutor cannot compile your program in the lab by typing make, you will receive **50%** of your final mark.

2. You must use version control from the get-go. This means that there must be a .git folder alongside the code in your project folder. A **10%** penalty apply should you fail to include a local repository.

3. You must provide a README file explaining what each file submitted does and how it fits into the program as a whole. The README file should **not** explain any theory that you have used. These will be used by the tutors if they encounter any problems.

4. Do **not** hand in any binary files. Do **not** add binaries (.o files and your executable) to your local repository.

5. Please ensure that your tarball works and is not corrupt (you can check this by trying to extract the contents of your tarball - make this a habit!). Corrupt or non-working tarballs will not be marked - **no exceptions!**

6. A 10% penalty per day will be incurred for all late submissions. No hand-ins will be accepted if later than 7 days.

7. **DO NOT COPY. All code submitted must be your own.** ***Copying is punishable by 0 and can cause a blotch on your academic record.*** **Scripts will be used to check that code submitted is unique.**