Agenda / Learning Objectives:

1. Run the following command and extract lab11.tar in your venus account (note the dot):

   cp ~ctse/cs211/lab10.tar . ; tar xvf lab10.tar

2. Check out the textbook resources inside **piazza**.

3. Use g++ -E option to run cmath_header.cpp and re-direct the console output to a file called cmath_header.

4. Glance through the manual page of "man g++" and read about the -c and -E options of g++.

5. Read the tips and pitfalls at the end on page 3.

6. Complete the two textbook exercises below:

Chapter 7 question 4. Hot Dogs

You operate several hot dog stands distributed throughout town. Definite a class named `HotDogStand` that has a member variable for the hot dog stand's ID number and a member variable for how many hot dogs the stand has sold that day. Create a constructor that allows a user of the class to initialize both values.

Also create a method named "JustSold" that increments the number of hot dogs the stand has sold by one. The idea is that this method will be invoked each time the stand sells a hot dog so that we can track the total number of hot dogs sold by the stand. Add another method that returns the number of hot dogs sold.

Finally, add a static variable that tracks the total number of hotdogs sold by all hot dog stands and a static method that returns the value in this variable.

Write a main method to test your class with at least three hot dog stands that each sell a variety of hot dogs.

Hint: Recall that static variables must be initialized outside of the class definition.

Chapter 11 question 4. Verify User.

Original question:

You would like to verify the credentials of a user for your system. Listed below is a class named `Security` that authenticates a user and password (note that this example is really not very secure, typically passwords would be encrypted or stored in a database):

```
class Security
{
public:
     static int validate(string username, string password);
};
// This subroutine hard-codes valid users and is not
// considered a secure practice.
// It returns 0 if the credentials are invalid,
//   1 if valid user, and
//   2 if valid administrator
int Security::validate(string username, string password)
{
  if ((username=="abbott") && (password=="monday")) return 1;
  if ((username=="costello") && (password=="tuesday")) return 2;
  return 0;
}
```

Break this class up into two files, a file with the header named `Security.h` and a file with the implementation named `Security.cpp`.

Next, create two more classes that use the `Security` class by including the header file. The first class should be named `Administrator` and contain a method named `Login` that returns true if a given username and password have administrator clearance. The second class should be named `User` and contain a method named `Login` that returns true if a given username and password have either user or administrator clearance.

Both the `User` and `Administrator` classes should be split up into separate files for the header and implementation.

Finally, write a main method that invokes the `Login` method for both the `User` and `Administrator` classes to test if they work properly. The main method should be in a separate file. Be sure to use the #ifndef directive to ensure that no header file is included more than once.

Simplified version for the lab exercise:

I have included the textbook solution for Administrator.h, Administrator.cpp and main.cpp in the lab11.tar file above. I would like you to come up with Security.h, Security.cpp, User.h and User.cpp. After that, compile all the codes by running **g++ *.cpp -o Driver** and test it out. You should get something similar to the following:

```
[ctse@venus ch11Prg4]$ ./Driver
Results of login:
User login for abbott: 1
User login for lynn: 0
User login for costello: 1
Admin login for abbott: 0
Admin login for lynn: 0
Admin login for costello: 1
```

From teacher's note for Absolute C++:

Key Points + Tips

Static Members. The keyword *static* is used in several contexts. C used the keyword static with an otherwise global declaration to restrict visibility of the declaration from within other files. This use is deprecated in the C++ Standard.. In Chapter 11, we will see that unnamed namespaces replace this use of static.

In a function, a local variable that has been declared with the keyword static is allocated once and initialized once, unlike other local variables that are allocated and initialized (on the system stack) once per invocation. Any initialization is executed only once, at the time the execution stream reaches the initialization. Subsequently, the initialization is skipped and the variable retains its value from the previous invocation.

The third use of static is the one where a class member variable or function is declared with the static keyword. The definition of a member as static parallels the use of static for function local variables. There is only one member, associated with the class (not replicated for each object).

Header Files and Implementation Files. Suppose a program has been split into several files. The functions defined in files which we usually name with a .cpp suffix . can be called in another file provided the functions are declared in the file where they are called, prior to being called. The code authors write header files to supply the declarations. The names of header files usually have a suffix of .h. This divides the code into the implementation file (which should not be available to the client) and the interface file (which the client uses to declare the functions prior to calls.) When used with makefiles, this separation also helps speed up compilation as only files that have changed need to be recompiled.

If the header file is intended to provide an interface and to document the functions in the implementation file, then the header comments should be provided in the header file as well as just the declarations. It is conventional to give the header file and the implementation file the same name apart from the suffix.

Using #ifndef. The #ifndef is one of several conditional compilation commands. A major use is to prevent multiple inclusion of header files. The commands are

```
#ifndef  PROTECT__DTIME_H
#define PROTECT__DTIME_H
// header file here
#endif
```

If the symbol PROTECT_DTIME_H is defined in the preprocessor symbol table, then the preprocessor will not insert into the temporary file any lines of code up to the next #endif and the compiler will not compile this code. (Failure to put in the #endif is a disaster.) If the symbol PROTECT_DTIME_H is not defined, the next line will define the symbol. The preprocessor will include all lines of code up to the next conditional compilation command. Then the compiler can compile these lines of code.

Pitfalls

Attempt to access non-static variables from static functions. Non-static class instance variables are only created when an object has been created and is therefore out of the scope of a static function. Static functions should only access static class variables. However, non-static functions can access static class variables.