SYDE 121
Lab Number 7

This lab contains **three** exercises. Only **two** exercises are to be submitted for grading. The **first** exercise is mandatory. Only **one of the second and third** exercises needs to be submitted.

The second exercise is more computer science oriented and has "bonus" marks associated with it, but these will require additional effort and ability. The third exercise is engineering-based (and is a bit easier), but does not have any bonus marks associated with it. Hint: The exercise that you do not submit would act as good practice in preparation for the final exam.

Separating the interface and implementation

Up to now, your programs have used only a single source file, which has contained all of your function prototypes, definitions, and declarations, as well as your main program. This lab introduces *modular programming*, which is the approach usually used in C++ programming.

The term *module* can be used to describe the combination of a .hpp (or .h or header) file, which contains the function *interface,* and its associated .cpp file, which contains the function *definition* or *implementation*. C++ programmers follow this convention by placing all structure (and class) declarations, together with all function prototypes, into a .h file. This file contains the function interface: everything a third-party user needs to see in order to use your code. The implementation (or definition) of each function is placed in a .cpp file. A third-party user would not need to see the details of your implementation in order to use the code effectively (assuming, of course, that your code is correct).

Placing all of the declarations in a .h file requires that you use an #include statement in each .cpp file that references those declarations. The statement will have almost the same form as when including files of the C++ standard library, except that quotation marks are used instead of angle-brackets. For example,

Use:

```
#include "myfile.h"        // my own header file
```

as opposed to:

```
#include  <cmath>          // a standard library header file
```

Since this file must be included in *each source file that references the declarations*, this introduces the danger of "multiple declarations"; that is, when processing the source files, the compiler encounters a declaration for the same type more than once, and does not know how to handle this potential ambiguity.

For this reason, it is common practice to "wrap" header files in compiler directives, as follows:

```
//
// usual header comments

#ifndef MYFILE_H
#define MYFILE_H

   // place your structure declarations and function prototypes here

#endif
```

Note the MYFILE_H is really just a variable name, and can be anything. However, it is usually selected to correspond to the filename. That is, just convert the entire filename to uppercase and then add '_H'. This is a standard technique in C++ programming.

Note that starting in this lab, we will not be using the naming convention used for the earlier labs. Normally, you will be given the filenames that should be used for a given problem. Please use these filenames.

Creating a Project In BloodShed

A *project* will contain all the necessary modules that you created to compile the code. As you learned before, the source files (*.cpp) are transformed to object code, and then the linker will group these into an executable (*.exe). You need to learn how to include the necessary files in the compile sequence.

1) Create a new project by selecting File->New->Project. Select an Empty Project and assign an appropriate project name. Save into a folder dedicated to this project.

2) Open a new file (File->New Source File) that will store the main function (*.cpp).

3) You can create new header files (*.h) and source files (*.cpp) as required. When doing so, make sure that you add the file to the project (you should be prompted for this). If this was not done or you

want to include an existing file, you can add a file by selecting Project->Add to Project.

4) You can view all files included in the project by looking under "Project" on the left hand side.

5) Under the compiler options, you can either compile the full project or individual files.

Exercise 1: 3D Coordinate Computations

Learning Objectives: Practice declaring and using structs in a program; practice passing structs to and from functions, using the const keyword, where appropriate; learn to deal with separate .h files and .cpp files for the interface and implementation, respectively; practice building your program incrementally.

Read this first

The Global Positioning System (GPS), a satellite positioning system developed by the US Department of Defense, has revolutionized positioning and navigation. In standalone mode, the system is able to provide real-time positions anywhere on earth, at any time of the day or night. GPS has found many uses amongst civilians and the private sector, too. In conjunction with a second GPS receiver, points can be positioned to millimetre level accuracy for geomatics- and surveying- related applications. Industry experts predict that positional information will become as commonplace as the time of day.

In this exercise, you are required to define a struct to represent 3D points positioned on the earth's surface by GPS, or some other positioning technique. You will then be required to perform a number of operations on these points.

What to do

1. Create a header file called coord.h.
2. In coord.h, write the declaration for a structure called Point3D, which contains the following members:
    - ID record of type int
    - X coordinate value of type double
    - Y coordinate value of type double
    - Z coordinate value of type double,
    - Classification record of type char, called order, representing the accuracy of the point (A = high accuracy, B = medium accuracy and C = low accuracy).

    "Wrap" the contents of the header file in the preprocessor directives described above, to prevent multiple declarations.
3. You have now declared the data *type;* you now need to check whether your declaration is correct. Create a file called gps.cpp. In main, declare two variables of type Point3D, called point1 and point2. These are now *instances* of the Point3D data type. (You will need at least two points to perform the computations required later in the lab.) Make sure that you have an #include statement which specifies "coord.h" — the compiler needs access to your structure type declaration in order to declare a variable of that type.

    Compile and run your program (it will not do anything at this point, but should be free of errors).
4. Write a function which prompts the user for values of the members of a Point3D variable. This function can be called from main to get user input for Point3D variables. The prototype for this function will look as follows:

    ```
    void get_point (Point3D &point);
    ```

    and should be placed in the coord.h file.

    Note that the variable called point passed to the function is of type Point3D, which you have defined, and it is passed by reference to the function, to allow its elements to be changed once the values have been input.

Important note: For efficiency reasons, it is common practice in C++ programming to pass aggregate (user-defined) data types to and from functions *by reference*. If the variable should not be modified by the function, the const modifier should be included to permit read-only access to the data members. This feature of C++ allows a very useful combination of efficiency and access.

5. Now create a file called coord.cpp, which will contain the implementation of each of the functions you will write. Once again, remember to #include your coord.h file. In the get_point( ) function, implement the code to prompt the user for the values of each member of a Point3D variable, store the values, and echo the input. Add the code in main to call this function for each of the variables you have declared. Test and debug this function.

6. Create a function to print a Point3D variable. Place the prototype in the coord.h file, add the implementation to coord.cpp, and add a call in main to test your new function. The prototype for the new function should look like:

```
void print_point( const Point3D &point );
```

Note the use of const in the function definition — a print function should not modify the variable that it is printing.

7. In a similar way, add functions to perform the tasks listed below, by adding the prototypes to the .h file, and each function implementation to the .cpp file. You will need to add function calls in your main program to test these functions. *You are strongly urged to write the code for one function at a time, and to test each function carefully before moving on to the next one.*

- Add a function called spatial_dist( ) to compute the spatial distance between 2 points, based on the formula:

$$d_s = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2 + (Z_2 - Z_1)^2}$$

- Add a function called plani_dist( ) to compute the planimetric distance between 2 points, based on the formula:

$$d_p = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}$$

- Add a function called ht_diff( ) to compute the height difference between 2 points (i.e., the difference in the Z coordinate values):

$$dh = Z_2 - Z_1$$

- Add a function called azimuth( ) to compute the azimuth (heading), which is measured clockwise from north, between 2 points, based on the formula:

$$\theta = \tan^{-1}\left[\frac{Y_2 - Y_1}{X_2 - X_1}\right]$$

Use the atan2 function in the cmath library, as it automatically computes the angle in the correct quadrant.

8. Complete the modifications to main so that:
   - the user is prompted to enter the values for *two points* using the get_point( ) function,
   - the parameters of each point are displayed on the screen using the print_point( ) function,
   - each of the above quantities are computed and displayed to the user, using your new functions.

   Apart from the print_point function, a*ll computed results should be output in main, not in the functions themselves.*

What to submit

Hand in and email your three source files, coord.h, coord.cpp and gps.cpp.

Exercise 2: Summing two numbers stored in arrays

Learning Objectives: Practice working with 1-d arrays and single character entry of data. Further practice using modular programming.

Read this first

This problem is taken from your Savitch textbook. An array can be used to store large integers one digit at a time. For example, the integer 1234 could be stored in the array 'a' by setting a[0] to 1, a[1] to 2, a[2] to 3, and a[3] to 4. However, here you might find it more useful to store the digits backward, that is, place 4 in a[0], 3 in a[1], 2 in a [2], and 1 in a[3].

In this exercise, you will write a program that reads in two positive integers that are 20 or fewer digits in length and then outputs the sum of the two numbers. Your program will read the digits as values of type *char* so that the number 1234 is read as the four characters '1', '2', '3', and '4'. After they are read into the program, the characters are changed to values of type *int*. The digits will be read into a partially filled array, and you might find it useful to reverse the order of the elements in the array after the array is filled with data from the keyboard. (Whether or

not you reverse the order of the elements in the array is up to you. It can be done either way, and each way has its advantages and disadvantages.)

Your program will perform addition by implementing the usual paper-and-pencil addition algorithm. The result of the addition is stored in an array of size 20, and the result is then written to the screen. If the result of the addition is an integer with more than the maximum number of digits (that is, more than 20 digits), then your program should issue a message saying that it has encountered "integer overflow". You should be able to change the maximum length of the integers by changing only one globally defined constant. Include a loop that allows the user to continue to do more additions until the user says the program should end.

What to do

As in Exercise 1, create three different source files for your program named sum.cpp, sumtwonums.cpp, and sumtwonums.h.

In this case, since you are entering single chars. Use the `get` cin member function to do this i.e., `cin.get( )`. To represent a longer integer number, it is not necessary to output the single char each time one is entered. Once the full number is entered and stored in a numerical array, then the entire number should be echoed to the user.

This problem is more challenging than Exercise 1. You are expected to complete all of the directions provided to you in the problem, and a few more listed below. About two-thirds of the marks allocated to this problem can be obtained by solving the minimum requirements. The remaining one-third can be obtained by implementing the advanced features and represent bonus marks for the assignment.

Minimum Requirements (two-thirds of the marks)

- create 3 separate source files (as in Exercise #1)
- code is neat, concise, and readable
- use of proper commenting
- read in two numbers one character at a time into numerical arrays
- echo both numbers to the user
- determine and display proper sum of the two numbers

Advanced Features (one-third of the marks)

- overflow check following the summation (do not exit program if overflow encountered)
- display numbers using commas e.g., 12,345,567
- only numerical chars are entered into the arrays i.e., for all data entered, only allow 0, 1, 2, … 9 to be entered into the arrays
- treat user-entered multiple zeros as zero i.e., '000' is displayed as '0'
- disregard leading zeros e.g., 00123 is displayed as 123
- allow user to loop through process as many times as desired
- allow user to quit at any time (even midway through data entry)
- act appropriately if user enters too many digits for a single number (exit is ok)

At the end of your header file, indicate which requirements/features work for your program and which features do not. Just list the above requirements/features at the end of your header file in a fully commented section and indicate which ones you were able to complete. This information will assist the TA when grading your submission.

Just meeting the minimum requirements will save time compared to implementing the advanced features. Note that there are different ways to implement the given problem. Ensure that your code is properly documented so that the TA is aware of how you solved the problem.

Hint: Design your code prior to going to the computer. Then, once you make a certain amount of progress on the computer, you will probably find that your original design is not working as planned. It is advisable to leave the computer and redesign your plan. This is not easy to do since the tendency is to stay at the computer. In terms of minimizing time spent solving the problem, knowing when to redesign the algorithm away from the computer is an asset.

What to submit

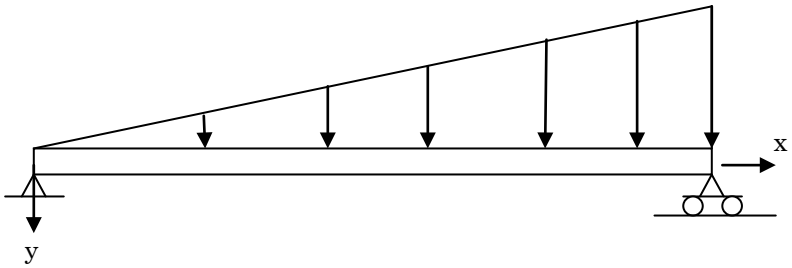
Hand in and email your three source files.

Exercise 3: Determining deflection and moment of a beam under loading

Learning Objectives: Practice working with 1-d arrays. Further practice using modular programming.

Read this first

Later in your SD curriculum, you will take a course in deformable solids. You will be asked to determine equations that describe the behaviour of various bodies under load. Unlike your first year statics course where rigid bodies are assumed, deformable solids better approximate the real world by recognizing that bodies under load will deflect. Beams are an important part of the engineering world and determining their deflection under certain loading conditions is a necessity. Here's an example:



For this beam, both ends are allowed to rotate (indicated by the triangles), however, the left end is fixed whereas the right end can move freely in the horizontal direction. The arrows pointing downwards describe the direction of load on the beam and the length of the arrow described the magnitude of that load at that point. In this case, the load continuously and linearly increases from left to right along the beam. This is helpful to understand, but what you really need to implement follows next.

The equation for the deflection (y) along the length of the beam (x) is (note that positive y deflection is defined as downwards):

$$y(x) = \left[\frac{2QL^4}{\pi^5 EI}\right]\sum_{n=1}^{\infty}\left[\frac{(-1)^{n+1}}{n^5}\right]\sin\left(\frac{n\pi x}{L}\right)$$

Although this seems like a messy equation, most of the terms are constants, where E is the "modulus of elasticity" (i.e. the material stiffness), I is the moment of inertia of the cross-section (i.e. the resistance to bending due to the cross-sectional shape), Q is the loading (as a function of length; in this case, the loading is triangular), and L is the length of the beam. 'y' represents the deflection as a function of 'x' (position along the length of the beam) and you will require an array to store this information. You can use the following constants in your header file:

```
const double ELASTICITY = 200e6;    // kiloNewtons / m^2
const double INERTIA = 8.3e-6;      // m^4
const double LOAD = 100.0;          // kN / m
const double LENGTH = 1.0;          // m
const double PI = 4.0*atan(1.0);
const double EPSILON = 1e-30;       // tolerance
const int STEPS = 10;
```

where "STEPS" is the number of discrete steps along the length of the beam where the deflection (y) and bending moment (M) are to be calculated. We calculate the displacement at discrete locations because we are not calculating a formula to calculate the deflection at any real number along the axis. If we select enough discrete points, then we have enough information to interpolate between these points.

You will have to use a tolerance (const EPSILON) to terminate the series summation.

The bending moment is given by:

$$M(x) = EI\frac{d^2y(x)}{dx^2}$$

What to do

As in Exercise 1, create three different source files for your program: beambend.cpp, beam.cpp, and beam.h. The main program will create the necessary arrays (initialize them) and pass them down into separate functions to determine the deflection (y) and the bending moment (M) along the beam at intervals of L/10.

You will need to calculate the second derivative. This is not difficult. In calculus, you learn about the continuous definition for a derivative. Here, since the computer works in a discrete environment, you need a discrete

approximation for a derivative. Here, we will use the forward discrete approximation:

$$y'[i] = (y[i+1] - y[i]) / d$$

where 'd' is the spatial displacement (namely the spacing along the length of the beam). Note that this just measures the slope of the deflection along the beam. The second derivative is determined by taking the derivative of the first derivative – you just have to create one function that determines the derivative and then apply it twice!

Don't worry about the physics and mathematics – these are concepts that are eventually covered in the SD curriculum.

You will probably want to create a generic function to display an array to the screen. This function would accept the size of the array as well as the array itself.

You should allow the size of the arrays to be a global constant; however, you should pass the size of the arrays into the functions as arguments along with the array itself.

Note that you will see some odd behaviour at the end of the beam due to the calculation of the derivatives at the end of the beam. Don't worry about this.

Optional

If you want to view the deflections (this is not part of the lab submission requirements), you can download the "output_to_file.cpp" program on the course website and integrate it into your program. This function accepts three parameters, the name of the array (just a string), the array itself, and the size of the array. The array will be sent to a file "out.txt" in the directory where the beam program is stored. You can take this output, cut and paste it to an Excel spreadsheet, and use Excel's plotting functions to plot the result. You can plot the deflection, its first and second derivative, as well as the bending moment. Note that there will be a numerical error near the end of the beam, but do not be concerned about this. Enjoy!

Note: you will soon learn how to send data to and from files.

What to submit

Hand in and email your three source files.

Due Date

All parts of this lab are due by Friday, November 4 at 6:00pm.

Reminders

Make sure that all your function prototypes are documented properly. See the Style Guide for an indication of how to do this.

Ensure that you are using `const` modifiers when necessary.