SYDE 121
Lab Number 7

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 .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 declarations**, 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 Dev-C++

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 project called gps.dev.
2. Create a header file called coord.h within the project.
3. 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.
4. 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).
5. Create a source file called coord.cpp in the project. For all of the functions described below, use this file to create the function definitions and use the coord.h to create the associated function declarations. Note, you will need to include the coord.h header file at the beginning of this coord.cpp file (`#include "coord.h";`). Also remember to include appropriate documentation (i.e. comments) in each of these files.
6. 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 function declaration 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.

7. 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.

8. 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.

9. In a similar way, add functions to perform the tasks listed below, by adding the function declarations 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.

10. Create a program file called gps.cpp for the main() function and the main part of your program (include directives, etc. Don't forget to #include "coord.h"). Within the main function:
    - 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.*

11. Create a "*README_gps.txt"* file, and create detailed instructions on how to run your *gps.cpp* program, including what input should be entered.

What to submit

Submit your three source files *coord.h*, *coord.cpp*, and *gps.cpp* and *README_gps.txt* files to the Lab Assignment 7 LEARN dropbox.

Exercise 2: Working with Pointers

Learning Objectives: Practice working with pointers and using pointers with regular variables and with structures.

Read this first

As we've learned in class, pointers hold a memory address of a variable or some other C++ data type, such as a structure or class.

Recall the syntax for using pointers on a variable is:

```
int *ptr;
int var = 0;

ptr = &var;

*ptr = 30; // This expression also sets var to 30,
           // via the pointer.
```

Using pointers with structs is similar, but has some different syntax for dereferencing the pointer (i.e. for accessing the data stored in the referenced memory address)

Using the Point3D struct that you defined in Exercise 1:

```
Point3D *ptr2;
Point3D point = {1, 30.0, 45.0, 40.0, 'B'};
                  // declare and initialize a point

ptr2 = &point;

ptr2-> order = 'C';  // The arrow (->) operator is used
                     // to deference a pointer to a
                     // struct. (note the difference
here
                     // between the normal syntax to
                     // access the struct's member
                     // variable, i.e. point.order
```

In order to better understand how pointers work and the relationship between the memory address they hold and the value (or data) stored in this memory address, this exercise will explore some basic pointer manipulation.

What to do

1. Create a program, called pointers.cpp (use only a single file for this exercise).
2. Create a structure called Circle, which has three member integer variables, centre_x, centre_y, and radius.
3. Write the main() function to reproduce the following output. The main() function should declare a variable of the Circle data type, along with a pointer to that variable. It should also declare a variable for the circle's circumference and a pointer to that variable. Note the blanks (___) in the sample output indicate missing data that will be filled in by your program or by the user. Also, you can create a helper function to read the input from the user and check that it is valid input (i.e. integers above 0).

```
Please enter the centre point and radius of the circle
(format: x y radius, in positive integers.)
__ __ __


The value of the variable circumference is ___
The address of the variable circumference is ___

The circle's location is (_, _)
The circle's radius is ___
The address of the circle struct is: ____
```

4. Create a *"README_pointers.txt"* file, and create detailed instructions on how to run your *pointers.cpp* program, including what input should be entered.

What to submit

Submit your *pointers.cpp* and *README_pointers.txt* files to the Lab Assignment 7 LEARN dropbox.

Due Date

All materials for this lab are due **Friday, November 7** by **1:30pm.**