CS11 Intro C++ Lab 1: Simple Units-Converter

For the next few assignments you will build a program to convert between different units. Maybe in a perfect world everyone would use metric, but we live in a world where mass can be measured in pounds (or pound-mass, more accurately), kilograms, or stone; distance may be in feet or miles or kilometers or furlongs; the list goes on.

Sometimes this can result in catastrophic failures. One of the most notable recent exmaples is the Mars Climate Orbiter, which most likely slammed into the Martian atmosphere and disintegrated, all because one software system was using English units while another was using metric units.

One simple approach to this problem is to create a "value-with-units" data type that packages the two components together, so that a value's units are always clearly indicated alongside the value. This information can then be used to verify that values are in the proper units, and even to perform simple conversions between different units where appropriate.

(Note that large systems usually don't take this approach, because the additional information can consume a significant amount of resources. Instead, the entire system will be designed to work with a single set of units. However, this doesn't always occur successfully.)

For this lab you will create the beginnings of a unit-conversion program, and then learn how to compile and test your program using the C++ compiler.

Important Note

The following sections specify filenames, class/method names, and types to use in your program, as well as specific input/output prompts. Please follow all of these specifications carefully so that we may test your program with automated scripts.

Values with Units

In a file `units.h` (note the file is all lowercase), create a class called `UValue`, short for a "united-value" or a "value-with-units." The class should have only two data members, the value itself (stored as a `double` so that we get lots of precision!), and the value's units represented as a C++ `string`.

Your class should declare a two-argument constructor that takes a value and some units, and stores those into the `UValue` object.

Additionally, declare two member functions: `get_value()` which returns the stored value, and `get_units()` which returns the units.

In a corresponding file `units.cpp`, implement the member functions for this class.

Immutable Classes

You might notice something interesting about this `UValue` class - it has no mutators. Once a specific `UValue` object is initialized, its contents cannot be changed. This is called an **immutable class**, and the individual objects are **immutable objects**. It is relatively easy to make an immutable class - simply provide no mutators.

This is a relatively common pattern for simple classes that hold values. It tends to be easier to write bug-free code with such objects, because it's simply not possible to have unexpected side-effects occur. (This can be particularly useful in multithreaded programs, although this unit-conversion program will not be multithreaded!)

Converting Between Units

Once you have gotten this simple class written, write a function to convert between units:

```
UValue convert_to(UValue input, string to_units)
```

You should declare this function in `units.h` and implement it in `units.cpp`. Note that this is not a member-function; it is not part of the `UValue` class! Rather, it operates on `UValue` objects, and it produces `UValue` objects.

For now this function will be very simple. In fact, it will not be very useful. We will correct this in the next lab, and create much more powerful unit-conversion functionality. But, for now, your function should do the following:

- If the input is in "lb" (pound-mass) and `to_units` is "kg" (kilograms), convert from pounds to kilograms by multiplying the input value by 0.45.
- If the input is in "gal" (gallons) and `to_units` is "L" (liters), convert from gallons to liters by multiplying the input by 3.79.
- If the input is in "mi" (miles) and `to_units` is "km" (kilometers), convert from miles to kilometers by multiplying the input by 1.6.
- If the function doesn't know how to convert the input value into the specified units, just return the input value back to the caller!

Feel free to add support for other units to this function, but you can probably see that this is not a very clever implementation. We will get much more sophisticated next time, and in fact you probably don't want to waste much time on this function since we are going to replace it pretty quickly.

Main Program

Once you have completed the above unit-representation and unit-conversion functionality, it's time to create a main program that we can run from the command-line! The excitement is almost overwhelming. Almost, but not quite.

In a file `convert.cpp`, write a `main()` function that does the following:

1. Prompt the user with the string `"Enter value with units: "`
2. The user should be able to enter something like "15 mi", and your program should read in these values and initialize a `UValue` object with these inputs.
3. Prompt the user with another string `"Convert to units: "`
4. The user should be able to enter something like "km", and your program should read this value into a C++ `string` object.
5. Next, your program should try to convert the input value into the specified units, using your `convert_to()` function. This will spit out another `UValue` object.
6. Finally, your program should report the results.
    - If the new `UValue` object has the specified units, your program should report: `"Converted to: [value] [units]"`. For example, with the above inputs, the program should produce `"Converted to: 24 km"`.
    - If the new `UValue` object doesn't have the specified units, your program should report the failure by outputting `"Couldn't convert to [units]!"`. For example, if you tried to convert "15 mi" to "m"

(meters), your program should output `"Couldn't convert to m!"`.

Since this is a `main()` function, make sure you "`return 0;`" at the end of it.

Commenting Your Code

It is extremely important to make sure that all of the code you write is well documented. This is true for two important reasons:

- Many projects involve multiple programmers, and if one programmer doesn't document the way their code works, it can be extremely difficult for the other programmers to figure out the code when this is necessary.
- Additionally, it is extremely common to come back to code that *you wrote yourself*, perhaps weeks or months earlier, and then you have to figure out what in the world you were thinking, or why you had to write the program the way you did.

Well documented code solves these problems very easily. It does require extra effort, but once you develop the habit, it becomes pretty easy to maintain a good quality bar. In fact, most of the time you will write the comments as you go, so that you don't have to go back and "fill in" very much commenting in your code.

In CS11 Intro C++, we expect you to include the following in your comments:

- Write a comment just before every class you declare, explaining the purpose of the class. Your comment doesn't need to be long, but it should completely and concisely describe the purpose of the class.
- Within your class declarations, write a comment just before every data-member in the class, describing the purpose of the data-member. Again, in many situations this will not be a long comment, but it should completely and concisely describe the purpose of the data-member, including any values with special meanings that the member can be set to.
- Write a comment just before every function and member-function you define, explaining the purpose of the function, along with any inputs that are invalid, or outputs that indicate special details. Again, this comment may not be long (particularly for simple accessors and mutators), but it should definitely be present.
- Within any function or member-function, make sure to explain any parts of the code that are particularly tricky or subtle. Don't just repeat the code; explain it.

  If the code is very simple, you don't need to write any comments; you can assume that anyone reading your code will be reasonably proficient with the language.

This set of requirements may seem burdensome at first, but after years of programming, you will probably find it is a good *minimum* level of commenting to strive for.

Also, be aware that **it is also possible to over-comment your code!** Commenting should enhance readability of a program, but you can definitely add so many comments that readability starts to be hindered. You will learn the right balance through experience.

Answer HW1 Questions

Part of this assignment is also to answer some simple questions in the file `hw1.txt`. They mainly focus on what compiler you are using for your assignments, and how you build your project. Shouldn't be too difficult.

Submitting Your Work

Once you have completed the above tasks, and you are reasonably confident that your code works as intended and is properly commented, you can submit your work through csman. Make sure to submit these files:

- `units.h`
- `units.cpp`
- `convert.cpp`
- `hw1.txt`

Assignment Feedback Survey

Please also complete and submit a feedback survey with your submission, telling us about your experience with this assignment. Doing so will help us to improve CS11 Intro C++ in the future.
