# CS11 Intro C++ Lab 2: Improved Units-Converter

This week we will extend our unit converter so that it has a cleaner and more extensible approach to unit conversions. You may have noticed that your `convert_to()` implementation was distinctly hard-coded; in fact, if you wanted to add more conversions, you would have to write more code in this function. Yick. What would would greatly prefer is for our conversion program to be *data driven* rather than being *hard-coded* with only a few conversions. In other words, we would like to have a table of conversions that we recognize, and when a conversion is requested, we can do a lookup in this table to perform the conversion.

Another limitation from last week is that we didn't have a good strategy for indicating when a unit-conversion couldn't be performed. This needs to be fixed as well.

# The `UnitConverter` Class

In your `units.h` file, declare a new class called `UnitConverter` that will keep track of all conversions we know how to perform. (You will want to declare it after the `UValue` class, since our converter will work with `UValues`.) This class will become the "brains" of our program. We can add specific conversions to this class, which will record them internally in a collection. Then we can ask this class to convert from one kind of unit to another, and if the class knows how to perform the conversion, it will return the converted results. Already this will be a big step forward from last week's implementation!

## Keeping Track of Conversions

Each conversion that your unit-converter knows how to perform will require three pieces of information: (*from-units*, *multiplier*, *to-units*). This specifies that if we have some number of the *from-units*, we can multiply it by *multiplier* to convert into the *to-units*. Here are some examples:

- ("mi", 1.6, "km") - 1 mi = 1.6 km
- ("lb", 0.45, "kg") - 1 lb = 0.45 kg

Inside your `UnitConverter` class, declare a nested `struct` to keep track of these details. You might call the struct a "`Conversion`", for example. Your `struct` can be very simple; it doesn't need to provide any member functions, for example.

Once you have a data type to keep track of conversions, you can add a `std::vector` data-member to your `UnitConverter` to record the collection of conversions that the object knows about. For example, if your nested struct is called `Conversion`, the vector might be declared as `vector<Conversion>`.

Of course, all of these details are private implementation details, so they should go in the `private` part of your class declaration. That said, be sure you document everything completely and concisely.

## Adding Conversions

Once you have a `UnitConverter` that can keep track of conversions, it's time to provide a way to add new conversions to the converter. Write a member function like this:

```
void UnitConverter::add_conversion(string from_units, double multiplier, string to_units)
```

This member function should do the following:

- Verify that the conversion doesn't already appear in the object! We don't want to add the same conversion multiple times. We don't just want to see if *from-units* appears in the vector; we need to see if there is already an existing that specifies both *from-units* and *to-units* in the same rule.

  If this case occurs, the method should throw an `invalid_argument` exception. (This exception type is declared in the C++ standard header `<stdexcept>`.) The message in the exception should say: "Already have a conversion from *from-units* to *to-units*", with the actual units in the message. You can initialize a `string` local-variable, and build up this error message in the local variable, before passing it to the exception initializer.

- If you get to this point, you know that the conversion isn't already in the object. Therefore, add the conversion to your vector of conversions: (*from-units*, *multiplier*, *to-units*)

- Also, if we know how to convert from miles to kilometers, we also know how to convert from kilometers to miles. Therefore, add a second conversion to your vector of conversions: (*to-units*, 1 / *multiplier*, *from-units*)

You can see how this member function can take care of a lot of work for us, so that the users of our class don't have to think very hard. They can just write lines like:

```
converter.add_conversion("mi", 1.6, "km");  // 1mi = 1.6km, and 1km = 0.625mi
```

The object will take care of the rest.

# Converting Units

Now let's migrate the `convert_to()` function into this class so that it can simply look up the conversion to use. Add another member function:

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

This member function should do the following:

- Try to find an entry in the object's list of conversions, with the same "from-units" as the `input` value, and the same "to-units" as the `to_units` value. If it finds one, it can multiply the input's value by the multiplier, and return a `UValue` object containing the calculated results.

- If the member function searches through the entire collection of conversions without finding anything, it should report failure by throwing an `invalid_argument` exception with the message "Don't know how to convert from *from-units* to *to-units*", with the actual unit types included in the message. As before, you will want to use a `string` local variable to build up this message, and then pass the value to the exception constructor.

# Main Program!

Now that we have a fancy new `UnitConverter` type to handle our unit conversions for us, we need to update our main program in `convert.cpp`.

## `UnitConverter` Initialization

Above your `main()` function, add a new function:

```
UnitConverter init_converter()
```

This function should declare a `UnitConverter` local variable, add a bunch of conversions to it, and then return the `UnitConverter` object to the caller. Add these conversions to your program:

- 1 mi = 5280 ft
- 1 mi = 1.6 km
- 1 ft = 12 in

- 1 in = 2.54 cm

- 1 lb = 0.45 kg
- 1 stone = 14 lb
- 1 lb = 16 oz

- 1 kg = 1000 g

- 1 gal = 3.79 L
- 1 bushel = 9.3 gal
- 1 ft^3 = 7.5 gal

- 1 L = 1000 ml

You may wonder why we are writing a separate function to do this initialization. A well designed program will separate its functionality into different sections such that each section addresses one concern. This principle is called **separation of concerns**. By separating the initialization code away from the main function, we can make changes to how the `UnitConverter` is initialized in the future, without affecting other parts of our program.

> NOTE: You don't have to worry about exceptions being thrown in this initialization code, since we are hard-coding the rules. If there is an exception in this code, it indicates a bug in our code!

## Using the `UnitConverter`

Once you have written your initialization function, you can use it in `main()`, like this:

```
UnitConverter u = init_converter();

... // Get the input value-with-units

UValue output = u.convert_to(input, to_units);
```

```
... // Output the results, or report an error in conversion
```

If the unit-conversion is successful, your program should print out the same results as last lab: `"Converted to: [value] [units]"`

However, this time if the unit-conversion fails, an exception will be thrown. Therefore you need to wrap the `convert_to()` line with a `try` / `catch` block that will report an error if an `invalid_argument` exception is thrown. This time your code should report the following:

```
Couldn't convert to [units]!
[message from the exception object]
```

> HINT: Put the successful-output code in the `try` block along with the attempt to convert. That way, if conversion fails, the successful-output code will not be run at all. Similarly, put the error-output code in the `catch` block, so that it only runs when there is an error.

Once you have completed all of this work, you should be able to compile and run your unit converter, and try any of the conversions that your program understands. You should also be able to convert in the opposite direction, and get an informative error when a conversion fails. Here is some example output:

```
$ ./convert
Enter value with units:  28 lb
Convert to units:  stone
Converted to:  2 stone

$ ./convert
Enter value with units:  14 stone
Convert to units:  lb
Converted to:  196 lb

$ ./convert
Enter value with units:  14 stone
Convert to units:  kg
Couldn't convert to kg!
Don't know how to convert from stone to kg
```

# Testing Code

You might notice that our unit-conversion program doesn't exercise all of the `UnitConverter` functionality. For example, we expect that no exceptions will be thrown when we add conversions, since we are hand-coding the list of conversions our program understands.

Because of this, it's good to exercise our code with a test suite that will tell us if there are any issues. We have provided one for you to use this week. You can download these files into your working directory:

- hw2testunits.cpp is a simple test suite for our converter
- testbase.h is a simple testing harness we use in the CS11 C++ tracks
- testbase.cpp is the implementation of the harness declared in `testbase.h`

Download these files into your local working directory, and you can compile them like this:

```
g++ -Wall -Werror units.cpp testbase.cpp hw2testunits.cpp -o hw2testunits
```

If your program compiles successfully, you can run it and see if all tests pass. If they do, you will have a higher confidence level that all of your code has been properly implemented.

# 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`

You do not need to submit the test code; we will test your program with a fresh copy of these files.

## 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.

---