CS11 Intro C++ Lab 3: Completed Units-Converter

Our units-converter is nearly complete! This week we will complete the implementation of all functionality in the program. Then, next week we can put a few polishing touches on the way our program is built and documented.

The features for this week are:

- Load our unit-conversion rules from a data file, instead of having them hard-coded in the program.
- Update our code to use references and the `const` keyword everywhere that is appropriate.
- Add one more clever feature to our program: If the user requests a conversion from unit A to unit B, and the program doesn't know how to do this but it knows how to convert from unit A to unit C, and then from unit C to unit B, we will use this to perform the conversion.

The last task is probably the most challenging one for this week, but we will get to it later.

Loading Unit-Conversion Rules

Last week we added an `init_converter()` function to our `convert.cpp` file, which is responsible for setting up a `UnitConverter` object with various conversions loaded into it. This week you should modify the function to take a single `string` argument, which is the filename of a file containing conversions for you to load. Next, modify the function to open this file and read the conversion rules, adding each one to the `UnitConverter` object.

> (Even though you are writing the code to support any filename, please call the file of conversion rules "`rules.txt`" so that it's easy for us to plug in a test file of conversions.)

This file should follow a very simple format, so that it is very easy to load. For example, given these conversions:

- 1 mi = 5280 ft
- 1 mi = 8 furlongs
- 1 ft = 12 in
- 1 in = 2.54 cm
- 1 m = 100 cm
- 1 km = 1000 m

The file should be formatted like this:

```
mi   5280    ft
mi   8       furlongs
ft   12      in
in   2.54    cm
m    100     cm
km   1000    m
```

Each line is of the form (*from-units*, *multiplier*, *to-units*). You can use stream I/O to read in the contents of this file. The specific spacing of the data won't affect how the code works, since C++ stream I/O consumes and ignores whitespace when reading values.

Your `init_converter` function should throw an `invalid_argument` exception if the specified file cannot be opened for some reason. Of course, if the file contains the same conversion rule twice, this will also cause the `UnitConverter::add_conversion()` function to throw an exception. Thus, you can see that

your `init_converter` will indicate very clearly if initialization cannot be completed for some reason.

Therefore, your `main()` function should also be updated to catch any exceptions thrown by the `init_converter` function, and gracefully report the error and exit the program. If an exception is thrown then your `main()` function should return 1 to indicate the failed execution. (Recall that the `main()` function is defined to return an `int` value; 0 indicates success, and nonzero values indicate that an error occurred.)

Once you complete this step, you should make sure your program still compiles and runs correctly. Never make too many changes to your code, without testing your work incrementally as you go. If you make a small change and it causes your code to stop working, you know exactly where to look to fix the problem. However, if you make a *lot* of changes and then discover much later that your program stopped working, you don't know where to start looking for the issue!

Function Arguments, References, and `const`

The next task is to update your source code to use references and the `const` keyword properly with all function arguments. This will avoid making unnecessary copies of arguments, and we will also be able to avoid the danger of unintended side-effects in our code.

These updates will focus entirely on function arguments. They won't focus on return-values, local variables, or data-members in classes.

- Review all functions in `convert.cpp`, making sure that all function arguments use `const` and references where appropriate. This should be pretty easy, since there are only two functions in this file. Don't forget that the C++ `std::string` type is a class.
- Review the `UValue` and `UnitConverter` classes in `units.h` and `units.cpp`, making sure that all function arguments use `const` and references where appropriate.

  Additionally, review every member function: if the member function doesn't change any state in the object that it is called on, then the function should also be marked `const` so that the compiler knows the function doesn't change the object.

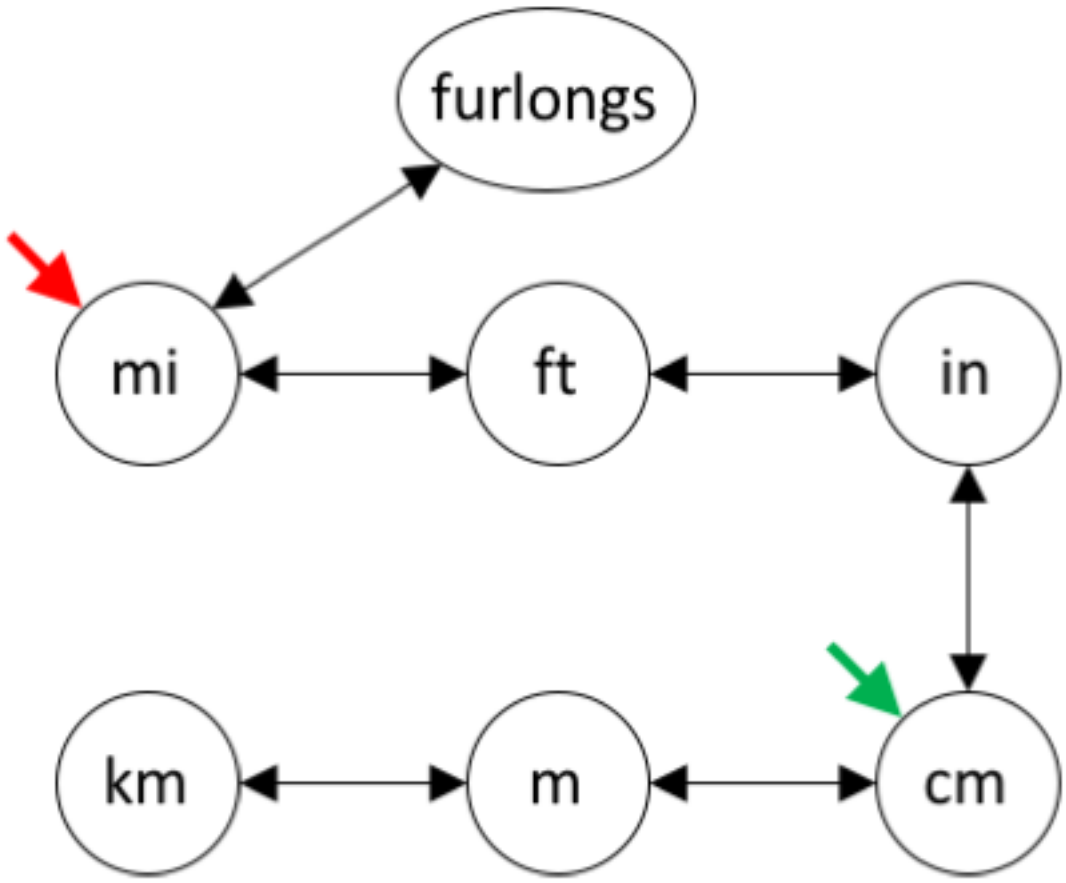
NOTE: You might recall that we designed the `UValue` class to be immutable, so that it's not possible to change a `UValue` object once it has been initialized. Even so, we should still use `const` properly so that we don't run into any surprises down the road. What kind of surprises? Well, if the `UValue` type were later changed to be mutable, then it would be necessary to go through all of your code and make sure you were using `const` correctly with it. By following the proper patterns from the beginning, you avoid having to make such changes later on.

As before, make sure that you test your program after completing these changes! In all likelihood you will run into various compiler errors as you make these changes, but once you are done, your program should be back in working shape once again.

Sophisticated Unit Conversions

The final task for this week is probably the hardest conceptually, but it will require a surprisingly small amount of code changes. Right now, we can convert from unit A to unit B, only if there is a conversion from A to B. However, if we have a conversion from unit A to unit C, and a conversion from unit C to unit B, we should also be able to convert from A to B. How do we build this?

Imagine we have these conversions, which are exactly the ones from earlier:



Length Conversions

Furthermore, let's say we want to convert from miles into centimeters. Clearly we don't have a conversion directly from miles to centimeters, but we do have a sequence of conversions that can take us from miles to centimeters through multiple steps.

We can implement a *recursive* solution to this problem. Recursion is a very powerful approach to problem solving, where you take a problem and solve it in terms of a smaller version of the same problem. For example, we clearly don't know how to convert *directly* from miles to centimeters, but we can try converting from miles to each thing we can convert it to, and then we can recursively try converting from that result into centimeters. If we keep doing this repeatedly - taking steps that we know how to perform, then recursively trying to solve the next step of the problem - we can eventually find the solution that gets us from miles to centimeters.

In other words:

```
convert_to(mi, cm)
  = convert miles to feet, then convert_to(ft, cm)
  = convert miles to feet, then convert feet to inches, then convert_to(in, cm)
  = convert miles to feet, then convert feet to inches, then convert inches to cm
```

You can see that each step of the way, we take the next steps we know how to take, and then try to solve the rest of the problem with recursion.

Avoiding Infinite Loops

There is one small issue though - each of our conversions goes both directions. We could easily get stuck in an infinite loop - converting from miles to feet, and then from feet to miles, and then miles to feet - forever! To solve this, our conversion function must remember what units it has seen along the way, so that it won't use those units again.

We can remember what units we have seen with another C++ Standard Library collection called `std::set`. This collection implements the notion of a mathematical set, where each element appears only once. This collection is also a template, so we can specify, an element-type of `string`, like this: `set<string>`. We will use this to tell if we have seen a given unit before.

- To use this data type, you must include the `<set>` header file.

- To add a value to the set, you can use the `set::insert(value)` member-function.

- There are several ways to see if a value is already in the set, but the easiest is to use the `set::count(value)` member-function. This will either return 0 (the value is not in the set) or 1 (the value appears one time in the set). A value can't appear more than once in the set, because it's a set!

Modify your `UnitConverter::convert_to(UValue, string)` function to take a third argument of type `set<string>`, representing the units that you have already seen. (Note that we are not specifying where `const` and references should be used in the arguments, so you can figure this out for yourself.) The `set<string>` argument will be updated as you go, to record the units that you have seen along the way. When your function recursively invokes itself, it will need to pass this set along, so that the nested invocations of the function will know what units have already been considered. (Hint: It doesn't matter if you pass this argument by-value or by-reference, so choose the one that makes the most sense to you.)

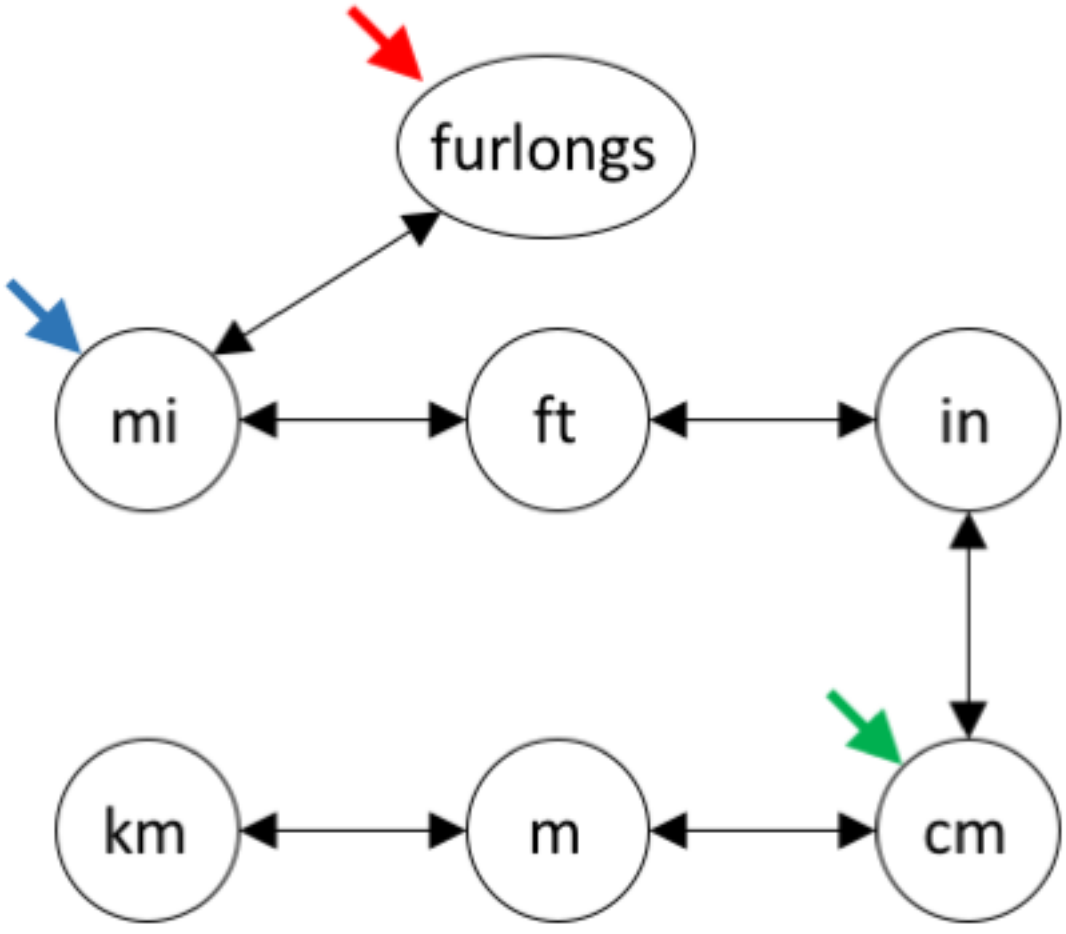
The logic of your `convert_to(UValue v, string to_units, set<string> seen)` will change to something like this:

```
convert_to(UValue v, string to_units, set<string> seen):

    Add v.units to seen, since we've seen this unit now.

    For each conversion c that we know about:

        If c tells us how to convert from v to to_units:
            Hooray, we are done!  Compute the result and return it to the caller.

        Else, if c has the same from-units as v, and we haven't seen c.to-units yet:
            Hmm, maybe we can use c to reach to_units eventually.

            Use c to convert v into a new value v' with units c.to-units.

            Then, return whatever convert_to(v', to_units, seen) returns.
```

Hopefully it's evident how the `convert_to()` function takes each step it knows how to take, and then recursively invokes itself to solve the rest of the problem. If a recursive invocation solves the problem, then that result is returned to the caller, terminating the recursion. Our `seen` collection prevents this recursion from continuing indefinitely.

Dead Ends?

There is one other wrinkle to solve: You can see from the picture that some conversions are dead ends! For example, what if our function tries converting miles into furlongs, and then gets stuck?!



Length Conversions

Well, just like before, we want our function to throw an `invalid_argument` when it fails to convert from the source units to the target units. At what point has the conversion failed? Looking at the above algorithm, you can see that if we iterate through *all* the conversions we know of, and still don't find any way to reach `to_units`, then the conversion has failed. So, we can update the above algorithm like so:

```
convert_to(UValue v, string to_units, set<string> seen):

    Add v.units to seen, since we've seen this unit now.

    For each conversion c that we know about:

        If c tells us how to convert from v to to_units:
            Hooray, we are done!  Compute the result and return it to the caller.

        Else, if c has the same from-units as v, and we haven't seen c.to-units yet:
            Hmm, maybe we can use c to reach to_units eventually.

            Use c to convert v into a new value v' with units c.to-units.

            Then, return whatever convert_to(v', to_units, seen) returns.

    If we reach here then the conversion failed.  Throw an exception.
```

Notice that we have added only one new step: If we iterate through all available conversions, and aren't able to do anything, then we should throw. Now our function will signal when the conversion has reached a dead end.

But, if we try converting miles to furlongs, and we don't see any way to get from furlongs to centimeters, does that mean we should give up?! No we shouldn't, since we can still try converting miles into feet, and then continue the conversion from feet instead.

Therefore, we need to make one more modification to this algorithm, to get to the final version:

```
convert_to(UValue v, string to_units, set<string> seen):

    Add v.units to seen, since we've seen this unit now.

    For each conversion c that we know about:
```

```
        If c tells us how to convert from v to to_units:
            Hooray, we are done!  Compute the result and return it to the caller.

        Else, if c has the same from-units as v, and we haven't seen c.to-units yet:
            Hmm, maybe we can use c to reach to_units eventually.

            Use c to convert v into a new value v' with units c.to-units.

            Then, return whatever convert_to(v', to_units, seen) returns.
            If this call throws an exception, catch the exception and ignore it;
            just go on to the next conversion and try to use it.

    If we reach here then the conversion failed.  Throw an exception.
```

In other words, if we make a recursive call to `convert_to()`, and that recursive call throws an exception, it doesn't mean that the entire conversion has failed; it just means that we couldn't find a conversion through that particular unit. So, we just catch the exception and then try the next conversion to see if that one will work. It's only when we have tried all possible options, and still failed to find a viable conversion path, that we should finally give up.

Modify your `convert_to()` function to operate in the way specified. Once you get it working, you should be able to convert between any pair of units that have a valid conversion path between them. (For example, you shouldn't be able to convert from miles to gallons, because that makes no sense. But, you should be able to convert e.g. between kilometers and inches, etc.

Preserving the `UnitConverter` Interface

You may notice that since we have added a third argument to `UnitConverter::convert_to()`, we would need to change all of the code that calls this function. What a drag. Fortunately we can avoid this by providing a second version of `convert_to()` which takes the same arguments as before, and calls the three-argument version, passing in an empty set:

```
UValue convert_to(UValue v, string to_units) {
    return convert_to(v, to_units, set<string>{});
}
```

(Again, we have not specified the use of `const` or references, so you can practice these skills.)

You should make a two-argument version of the member function like this, so you don't have to make any changes to how the `convert.cpp` code invokes unit conversions. Additionally, it will allow the provided test code to work as well.

All Done... For Now!

Once you complete all of the above tasks, you will be finished with the functionality of your unit-conversion utility. You should make sure to test your program thoroughly to ensure it works properly, and make sure that all of your work is well documented.

Again we have provided some test code to exercise the `UnitConverter` functionality. You can download these files into your working directory:

- hw3testunits.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++ -std=c++14 -Wall -Werror units.cpp testbase.cpp hw3testunits.cpp -o hw3testunits
```

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.
