CSE250, Spring 2014 Assignment 5 Due Fri. Mar. 7 & 14

This assignment is for **online submission only**, by **11:59pm Fri. Mar. 14, *with** a *checkpoint deadline* of **11:59pm Fri. Mar. 7** on *just the template conversion of your deque and peek classes*, to ensure that you have attempted them in a timely fashion.

Submit by Mar. 7, two files: `PeekDequeTemplateNNN.h` with your initials in place of `NNN`, and likewise `PeekClientNNN.cpp`, which will overwrite your previous version. If you use `StringWrap` already rather than `string`, please #include `StringWrap.cpp` as well as `StringWrap.h` in your client file, and submit them too *if you have made any alterations* to them. No makefile needed! *(changed to align with labs this week)*

Submit by Mar. 14: `PeekDeque3NNN.h` with your deque and peek-deque classes as template classes only, optionally revised `StringWrap.h` and `StringWrap.cpp` classes (no initials, and no initials in *any* class name), file `ChainClientNNN.cpp` with `ed1` and `main` and any helper functions, and `ChainMakeNNN.make` as a make-file for your project. *(no change).*

In-between, **Prelim I** is being held **Monday, March 10** *in class period*. It covers the text up through Ch. 2 and homework through Assignment 4.

"Mini-Project": Growing Word Chains

A *word chain* is a sequence of words in which each adjacent pair have edit-distance one. They are most familiar as word puzzles that use only the equal-length case of edit distance. For example, `hate` can be transformed into `love` via the chain

`hate-have-gave-give-live-love.`

Usually the puzzle is to find the shortest chain. If the word `lave` (meaning to wash) is in your dictionary, then you can save two steps via `have-lave-love`. Our goal, however, will be the opposite: to form chains that are as long and "interesting" as possible, while taking words one-at-a-time from a source like *Hamlet* or the U.S. Constitution.

Here we are helped by the ability to form more-general chains that add or subtract a letter, using all cases of edit-distance one. An example of a chain that grows at each step is

`a at ate rate irate pirate pirated.`

A sentence from which it could be grown is, "A VP at Microsoft ate at a faster rate while irate at pirate hackers and pirated software." This particular chain only adds letters at the ends. An example that uses the letter-in-the-middle case of edit distance, and that meanders up and down, is

```
glanced lanced laced lace race ace aced ached ache Ach! each reach preach
preachy
```

Note that especially when reading from sources, we will want to eliminate punctuation from words and treat all letters as lowercase, whether they are or not. The C++ `string` class

does not have methods to do this directly, so we will want to augment it. The deque data structure is natural for forming these word chains. We would *like to* be able to insert words in the middle, but we only have quick access to each end, and we picture the word stream as coming really fast. So for this task we will be content with the front-and-rear methods of our deque structure. We will generalize our deque classes so they can work with any augmented classes we might desire.

The above examples are nice in not repeating any words, and in having some long words. If we want to maximize the chance of getting long words in our chains we can try *not* attaching a word at either end that steps down the length. This, however, frustrates getting long chains because short words are usually easier to attach to. But if we want long chains, we will have to allow words to repeat. The fact that equal words are *not* edit-distance neighbors prevents immediate repetition. Whether to allow going back to the same word, as in `hip hop hip hop hip ...`, is another issue. Preventing this would need methods to peek not just at the front and rear word, but at the second word on either end. This is still OK even when dealing with a rapid stream, but iterating thru a whole chain each time just to prevent all repetition has the same time-crunch problem as inserting in the middle. We *can* cut down on the worst repetition by disallowing words of length 1 or 2 altogether—so that the "pirate" example above would have to start with "ate," but the example with "ace" and "Ach" is still good. Then, what kind of nice chains can we find in various texts?

Task Statement

Read words one-at-a-time from a text file given as a command-line argument. For each word, try to attach it at the front or back of any chains you have going already, and if you can't (or don't want to!—e.g. if it's a shorter word), start a new chain with that word. At the end of the file (or after a fixed number n of words, using a second command-line argument `n` intended mainly to help in debugging), report the longest chain(s) found, and one(s) with the longest word. Long and short example files such as `Hamlet.txt` and `Gettysburg.txt` are in the `Java2C++` directory, as are the `.cpp` and `.h` files for the `StringWrap` class.

Optionally (some chance for extra credit here), report chains that give a lot of different words, or are best according to some formula you make up that combines the niceness criteria discussed above. Or allow adding prefixes and suffixes of arbitrary length (but not taking them away?), so that e.g. `pirated` could go to `respirated`!

Some report questions will also be included for the 14th. Generally it is desired to use methods that run in good time for the length of *chains*, but the length of individual *words* for your `ed1` etc. methods is not so important.

Coding Details, and What's Due When

We already have a fast implementation of the deque data structure, the edit-distance-one function, and experience with one strategy in `main` for adding words to a chain and checking they are OK. To take advantage of existing code in a "wrapper class" called `StringWrap`, we want to change our deque to work with it instead of `string` literally. Better yet, we will make those classes work the same way with *any* type `T`—or at least any `T` that has a string conversion and a zero-parameter constructor. We will do this by converting `StringDeque` and `PeekDeque` into C++ *template classes* `Deque<T>` and `PeekDeque<T>`. Along with the necessary `friend` line (now we desire it instead of using `protected`), the exact syntax involved in declaring these classes is (overleaf):

```
template <typename T>
class PeekDeque;      //MAY be necessary on your system, as on timberlake

template <typename T> //REQ: T defines T() and [specify operator<< or str()]
class Deque{
 public:
   //class Deque<T>;  //error because confused with inner class.
   friend class PeekDeque<T>;

...
...

template <class T>
class PeekDeque : public Deque<T> {  //NO "PeekDeque<T>" here, nor above!
 public:
   PeekDeque<T>(int guarcap)         //but PeekDeque<T> in constructor is OK
    :
      Deque<T>(2*guarcap)            //ETC.---rest is for you to work out
```

Everywhere `string` occurred in your original versions, now they will say `T`—inside `vector<...>` which is already a template class, as argument type of your `push---` methods and return of your `pop---` methods and any item-getters you wrote. Everywhere, that is, except in `toString()`, which will still produce and return a `string`.

Our C++ compilers also now conform to two extra bits of mandated syntax inside template classes. One is that whenever the template parameter (such as `T`) appears inside angle brackets in a declaration as part of a type name (other than a method parameter), one must put `typename` before the whole declaration. The text notes this on page 281, but it shouldn't be needed in this code. The newer one affects derived template classes like `PeekDeque<T>` from `Deque<T>`: *every* use of a base-class member function or field needs to be qualified either with `this->` or with the base-class template name and `::` Here is an example; you need not have it exactly like-so but will probably have something similar:

```
   virtual void pushRear(T newItem) {
      if (this->full()) {    //or this->isFull()
         cerr << "Error: PeekDeque object is full" << endl;
         cerr << this->toString() << endl;
      } else {
         this->elements->at(++(this->rearItem)) = newItem;
         //or could do: Deque<T>::pushRear(item); to invoke super's method
      }
   }

   ...

   virtual void moveFrontward() {
      peekIndex--;  //what if it's now < 0?  Hmmm.....!
   }
```

Note the option to use the latter form with `::` in `full` in order to call the shadowed base-class method anyway—C++ always does this in place of Java `super`—so we wouldn't need the `this->` qualifier there. Note also that we did **not** use it with the `peekIndex` field in `moveFrontward()` because it is in the **same** class. Also note that especially when things are combined with arithmetic operators like pre-decrement, it is a good idea to put parentheses around things like `this->rearItem`. *Okay, the extra* `this->` *is a horrible kludge with awful syntax even given you have to have it, and while your home compiler may not require it yet, on* `timberlake` *it is needed, and it never hurts...*

Then revise your `main` from Assignment 4 to declare and initialize a `PeekDeque<StringWrap>*` pointer and use `StringWrap` where needed. Read in words using the idiom (compare text p48)

```
while((*INFILEp) >> word) {   //maybe add && <another test> to loop
```

You have to call the explicit constructor `StringWrap(word)` to convert what you read into a `StringWrap` object. For calls to `ed1` you can leave `word` as a `string` (i.e., you're not expected to rewrite `ed1` to use `StringWrap`), but to convert a `StringWrap` object `sw` to pass as an argument to `ed1` or to print out, it is recommended to call the method `str()` (rather than define an `operator string` or etc.). (We have made this name different from the deque's own `toString()` method to help you keep things straight initially.)

First convert the same `main` you had for Assgt. 4 to this new notation, and show it working along with `PeekDequeTemplateNNN.h` doing the same task as on Assgt. 4. (Alternatively you may revise `PeekClientKWR.cpp`, which is now available in `../Java2C++/` as the Assgts. 2–4 code answer key, and/or use the given `PeekDequeNonTemplateKWR.h` rather than your own code.) *This is all that is due on Fri. Mar. 7.* The code may all be in one file `DequeTemplateTestNNN.cpp` (the make-file isn't until the final submission).

Recitations in the week of Mar. 3–7 will give hands-on help with your template classes, and getting them to work with the other things you want to do in `main`. Preparing for this lab help is why we need the checkpoint submission. The whole project is due on **Fri. Mar. 15, 11:59pm**, with file names specified above. The grading is:

- 24 pts. for complying with the checkpoint submission—of which (only) 6 pts. is for code that already compiles. We will give the other 18 pts. so long as there has been clear effort to follow all the directions above in a timely manner.
- 60 pts. for the final code.
- 18 pts. for reporting on the strategies for growing word chains that you devised and the chains you found, for 102 points. Followup questions *may* bring it up to 120.

Some supplemental homework questions on running time (with Big-Oh and Theta notation), and/or logical code requirements may also reference this assignment.

Using the `gfilt` utility in place of `g++` on `timberlake` may greatly help in understanding error messages with templates, and it is highly recommended now. It may be usable even without typing the full path `util/bin/gfilt` as indicated on the course webpage. Remember to hit the spacebar if it displays more than a screen, and to hit 'q' to quit at the end.