Exam Facts:
When: Friday, February 9th from 1:30 - 2:50 p.m.
Where: Building 380, Room 380Y (same building as lecture, but different room)

Coverage
The exam is open-book, open-note, closed-electronic-device. We will not be especially picky about syntax or other conceptually shallow ideas. We are simply looking for a clear understanding of core programming concepts. As needed, we will present the prototypes of functions and methods we expect you’ll need.

Writing code on paper in a relatively short time period is not quite the same as working with the compiler and a keyboard. We recommend that you practice writing out solutions to these practice problems—starting with a blank sheet of paper—until you’re certain you can write code without a computer to guide you.

The practice midterm draws its problems from the first midterm I gave in CS106X two years ago this quarter, though that was a two-hour midterm (given on a Tuesday evening), and yours will be designed to be taken in class in 80 minutes. Understand that I’m under no obligation to imitate the format of this exam, though. I’m simply presenting this practice midterm to give you a sense of what types of problems have been given on CS106X midterms in previous years.
Summary of Relevant Data Types

bool isalpha(char ch);
bool isdigit(char ch);

class string {
    bool empty() const;
    int size() const;
    int find(char ch) const; // returns string::npos on failure
    int find(char ch, int start) const; // returns string::npos on failure
    string substr(int start) const;
    string substr(int start, int length) const;
    char& operator[](int index);
    const char& operator[](int index) const;
};

string trim(const string& str);

class Stack {
    bool isEmpty() const;
    void push(const Type& elem);
    const Type& peek() const;
    Type pop();
};

class Map {
    bool isEmpty() const;
    int size() const;
    bool containsKey(const Key& key) const;
    Value& operator[](const Key& key);
    const Value& operator[](const Key& key) const;
};

class Set {
    bool isEmpty() const;
    int size() const;
    void add(const Type& elem); // operator+= also adds elements
    bool contains(const Type& elem) const;
};

class Lexicon {
    bool contains(const string& str);
    bool containsPrefix(const string& str);
    void add(const string& word);
};
Problem 1: Shunting-Yard Algorithm [10 points]

The **shunting-yard** algorithm is an algorithm one can use to convert traditional, infix arithmetic expressions to postfix ones. For simplicity, we’ll assume the operands are all single digit numbers and that the only two arithmetic operators are + and *. * is of higher precedence than +, but parentheses can be used to override that. Because the operands are constrained to be single digit, expressions can be codified as strings, as with "(3+1)*(8+2)", no extraneous spaces. Like the RPN algorithm discussed in lecture, shunting-yard is stack-based.

Here’s a short program and some sample input and output to illustrate what needs to happen:

```cpp
int main() {
    while (true) {
        string infix = trim(getLine("Enter an infix expression: "));
        if (infix.empty()) break;
        cout << infix << " --> " << infixToPostfix(infix) << endl;
    }
    return 0;
}
```

Enter an infix expression: 3+6
3+6 --> 36+
Enter an infix expression: 1+3*3
1+3*3 --> 133+
Enter an infix expression: 1+(3*3)
1+(3*3) --> 133+
Enter an infix expression: (1+3)*3
(1+3)*3 --> 13+3*
Enter an infix expression: 1+3+1*3
1+3+1+1*3 --> 13+1+13+
Enter an infix expression: 4*(1+3)*((6+4)+2)
4*(1+3)*((6+4)+2) --> 413+*64+2++
Enter an infix expression: (3+1)*(8+2)
(3+1)*(8+2) --> 31+82++
Enter an infix expression:

Here’s the recipe your **infixToPostfix** implementation will follow:

  - for each ch in infix:
    - if ch is a digit
      - append ch output string
    - if ch is an open parenthesis
      - push that open parenthesis onto stack
    - if ch is a close parenthesis
      - pop operators off stack, appending to the output string
      - repeat until open parenthesis encountered, which should be discarded
    - if ch is an operator
      - while stack top is op of equal or higher precedence, pop and append to output
      - push ch onto stack
  
  - once infix has been scanned, drain stack, appending popped items to output

Using the next page to implement **infixToPostfix**, and assume the supplied expression is well formed so that no error checking is required.
Problem 1: Shunting-Yard Algorithm [continued]

    static string infixToPostfix(const string& infix) {
Problem 2: Autocorrect [10 points]

We all know that when our big thumbs type out big words on our smart phones, we mistype and spell some words incorrectly. We also know the phone itself presents one or more words it thinks we meant to type. If, for instance, we’re texting and type out "tounf", the phone might suggest "young", because it knows that "tounf" isn’t a word but that ’t’ is right next to ’y’ and ’f’ is right next to ’g’ on the keypad. This particular suggestion required two changes, but there aren’t any words in the English language that are one character away from "tounf", so "young" is a reasonably good suggestion (as are "round" and "found").

Implement the recursive ls function (ls is short for listSuggestions), which given a string, lists all of the words in the English language that require no more than a threshold number of substitutions. Your implementation should code to the following prototype:

```cpp
static void ls(const string& str, const Lexicon& english,
               const Map<char, string>& alternatives, int maxChanges);
```

str may or may not be a word in the English language, but if it is, it should be printed. Other words in the language should be printed if they require at most maxChanges letters to be replaced by their neighbors. alternatives has 26 keys—one for each lowercase letter—and each maps to a string of all of the keyboard letters immediately adjacent to it—that is, what we consider reasonable alternatives. For example, ’g’ maps to "tyfhcvb", because those seven letters represent what a big thumb might have intended to hit when it tapped the ’g’.

Use the next page to present your implementation, and feel free to tear this page out so you can easily refer to it.
Problem 2: Autocorrect [continued]

    static void ls(const string& str, const Lexicon& english,
                    const Map<char, string>& alternatives, int maxChanges) {

Problem 3: Regular Expressions and String Matching [20 points]

A regular expression—or regex, for short—is a string used to pattern match words in the English language. The simplest regular expressions consist of just lowercase letters, but they’re also allowed to contain one or more character sets like \([a-z]\), and the presence of \([a-z]\) in a regular expression matches any lowercase letter. Here’re a few examples of regular expressions and the English words that match them:

\[
\begin{align*}
\text{regex} & \quad \text{matches} \\
\text{and} & \quad \text{and} \\
[a-z]lur & \text{blur, slur} \\
w[a-z] & \text{wild, wile, will, wily} \\
m[a-z][a-z]m & \text{maim, malm, marm, mumm} \\
x[a-z][a-z][a-z]x & \text{xerox} \\
[a-z]x[a-z] & \text{axe, exo, oxo, oxy}
\end{align*}
\]

The notion of a character set can be generalized to specify one or more smaller ranges to represent sets of lowercase letters, as with:

\[
\begin{align*}
\text{character set} & \quad \text{possible characters} \\
[a-g] & \text{abcdefgh} \\
[c-gm-w-z] & \text{cdfgmwxz} \\
[aeiou] & \text{aeiou} \\
[x-za-bp] & \text{abpxyz}
\end{align*}
\]

Note that isolated characters can sit among zero or more ranges to compactly express a small set of characters, as I do with the three of the four sample character sets above. This notation allows us to match a more constrained set of English words:

\[
\begin{align*}
\text{regex} & \quad \text{matches} \\
m[aeiou][x-z] & \text{max, may, mix, miz, moy, moz, mux} \\
z[a-cor-z][a-gkn-p] & \text{zag, zap, zoa, zoo} \\
[a-c][d-g][h-m][n-q][r-z] & \text{adios, agios, aglow, below}
\end{align*}
\]

Finally, an asterisk (i.e., one of these things: \('*'\)) can follow any character or character set as an instruction that the single character or character set preceding it can be skipped and go unmatched, be matched exactly once, or be matched an arbitrarily large number of times.

What can regexes look like now, and what strings do they match? Here are some examples:

- \(aa[a-z]^*\) matches all those words that begin with aa, including aa, aah, aahed, aardvark, aardvarks, aarti, and aasvogel. The \([a-z]^*\) portion of \(aa[a-z]^*\) can match the empty string, a single letter, or an arbitrary string of length 2 or more.
• \([a-z]*zz[a-z]*/ matches all of those words that contain a zz somewhere, including buzz, jacuzzi, pizzelle, sizzle, spazzing, zyzyvas, and zzzs.

• \([a-z]*zz[a-z]*zz[a-z]*\) matches all of those words containing two independent double z’s. This list is pretty small, but it’s nonempty! It matches exactly 11 words, and bezzazz, pizzazz, and razzamatazz are among them.

• \([a-g]*\) matches all those words that can be formed using just the first seven letters of the alphabet, including begged, cabbage, deface, defaced, feedbag, and gaffed. Musicians love these words, because they can be formed using just the notes of a C major scale.

• \([aeiou][aeiou][aeiou][aeiou]*/ matches all of the English words of length 3 or more that contain only the five principal vowels.

• \([a-z]*a[a-z]*e[a-z]*i[a-z]*o[a-z]*u[a-z]*y[a-z]*\) matches the six words that contain all six vowels (this time counting y) where a, e, i, o, u, and y appear in that order. Congratulations to abstemiously, adventitiously, autoeciously, facetiously, halfseriously, and sacrilegiously for being part of this distinguished set.

For this problem, you’ll be led through the decomposition of a recursive function called \texttt{matches} that decides whether a regex matches a string of lowercase letters (presumably a word in the English language). Over the course of the problem, we’ll confirm you’re fluent with C++ strings and pass-by-reference.
a) [5 points] Your implementation of **matches** should benefit from a helper function called **expand**, which takes a single character set and returns a sorted string of all of the lowercase letters it expands to, as with:

<table>
<thead>
<tr>
<th>set</th>
<th>expand(set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a-g]</td>
<td>abcdefg</td>
</tr>
<tr>
<td>[x-ya-g]</td>
<td>abcdefgxy</td>
</tr>
<tr>
<td>[a-empw-z]</td>
<td>abcdempwxyz</td>
</tr>
<tr>
<td>[aeiou]</td>
<td>aeiou</td>
</tr>
<tr>
<td>[a-ea-ed-fa-eeeee]</td>
<td>abcdef</td>
</tr>
</tbody>
</table>

Your implementation should be able to handle redundancies like those you see in the last example above, and the string of lowercase letters returned should be sorted in lexicographic order. You should assume that the first character is always ' [ ', the last character is always ' ] ', there's at least one character between the ' [ ' and the ' ] ', and that the character set identifies only lowercase letters and is otherwise well formed. This part doesn't involve recursion, but it will test your ability to manipulate strings.

Use the rest of this page for your implementation:

```cpp
static string expand(const string& set) {
```
b) [5 points] Your implementation of `matches` will also benefit from a second helper function called `split`, which takes a nonempty regular expression and pulls off the portion that might be matched by a word’s first character. Here’s the interface you’re coding to:

```
static void split(const string& regex, string& first,
                  bool& starred, string& rest);
```

Assuming that `first` and `rest` are `string`s and `starred` is a `bool`, the following illustrates how `first` and `rest` would be populated for the provided regexes:

<table>
<thead>
<tr>
<th>regex</th>
<th><code>split(regex, first, starred, rest)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>awxyz</td>
<td><code>first</code> gets &quot;a&quot;, <code>starred</code> gets false, <code>rest</code> gets &quot;wxyz&quot;</td>
</tr>
<tr>
<td>[ae]*w</td>
<td><code>first</code> gets &quot;[ae]&quot;, <code>starred</code> gets true, <code>rest</code> gets &quot;w*&quot;</td>
</tr>
<tr>
<td>z</td>
<td><code>first</code> gets &quot;z&quot;, <code>starred</code> gets false, <code>rest</code> gets &quot;&quot;</td>
</tr>
<tr>
<td>z*</td>
<td><code>first</code> gets &quot;z&quot;, <code>starred</code> gets true, <code>rest</code> gets &quot;&quot;</td>
</tr>
</tbody>
</table>

To be clear, `starred` is populated with `true` if any only if the leading portion placed in `first` is optional and repeatable, and `rest` is populated with everything beyond `first` and, if present, the companion *.

Use the rest of this page to present your implementation, and don’t expand the character sets (you’ll do that part for part c). There’s no recursion here either, but it exercises your fluency with C++ strings and your understanding of pass by reference.

```
static void split(const string& regex, string& first,
                  bool& starred, string& rest) {
```
c) [10 points] Using the `expand` and `split` functions you’ve already implemented, present your implementation of `matches`, which uses recursive backtracking to decide whether the supplied regex matches the supplied word. Because backtracking is required, you should only make as many recursive calls as needed in order to produce a `true` or `false`. This is your chance you convey your understanding of recursive backtracking, pass by reference, and string manipulation, all in the same problem.

```cpp
static bool matches(const string& regex, const string& word) {
```