

Section Handout

Problem 1: Using the Scanner and Stack classes

```
<html><b><i>CS106 rules!</i></b></html>
```

Web browsers use stacks to track html tags such as ``, `<i>` or `<html>`. Every html tag must be matched by an equivalent closing tag—``, `<i>` or `<html>`. Microsoft is looking for programmers to help implement this feature in the next version of Internet Explorer and you, armed with your newly acquired knowledge of classes, decide to volunteer for the job. Using the **Scanner** and **Stack** classes to write the following function:

```
bool IsCorrectlyNested(string htmlStr);
```

You can assume that all the tags in the html string will be correctly formed. That is, once you see an angle bracket, it will be followed by the remainder of a complete and well-formed tag (So, nothing like "`<<html>`").

Problem 2: The Writing Center

In an effort to improve your English Literature grade, you've decided to write a program that reads in one file (presumably your first draft) and writes the improved version to another. Feedback on prior papers suggests you'd do better to cut out the adjectives and to use stronger verbs. So, you'd like to automate the rewrite process. (Or rather, "you'd *propose* to *automatize* the rewrite process." Wow! That is SO MUCH CLEARER!)

Write a function called **ImprovePaper**, which reads your first draft from the specified **ifstream** and writes the "better" version to the specified **ofstream**, using the supplied thesaurus (which maps words to synonym sets), and removes all adjectives and replaces all verbs with the longest synonym in the corresponding synonym set. Fortunately, someone else gave you code that magically tells you whether a word is being used as a verb or an adjective. These functions have the following prototypes:

```
bool isAdjective(string word);  
bool isVerb(string word);
```

You should assume that the files have been properly opened and that no error checking (beyond the end-of-file check) need be done. You should assume that the supplied thesaurus is fully populated, but you should **not** assume that it contains all the verbs in your paper. You should preserve spacing, and you shouldn't worry about any remnant punctuation that remains because a series of adjectives were removed. And as far as the verbs are concerned, you should assume the longer the verb, the stronger it is. If the

original word happens to be longer/stronger than all of its synonyms, then retain the original word.

```
void ImprovePaper(istream& in, ostream& out, Map<Set<string> >& thes)
{
    out << "Here's the better paper" << endl; // write to out as you would to cout
    out << "-----" << endl;
    // continue with your own code...
```

Problem 3: Blood Types

Every person's blood has 2 markers called ABO alleles. Each of the markers is represented by one of three letters: A, B, or O. That means there are six possible combinations of these alleles that a person can have, each of them resulting in a particular ABO blood type for that person.

Combination	ABO Blood Type
AA	A
AB	AB
AO	A
BB	B
BO	B
OO	O

Likewise, every person has two alleles for the blood Rh factor, represented by the characters + and -. Someone who is "Rh positive" or "Rh+" has at least one + allele, but could have two. Someone who is "Rh negative" always has two - alleles.

The blood type of a person is a combination of ABO blood type and Rh factor. The blood type is written by suffixing the ABO blood type with the + or - representing the Rh factor. Examples include A+, AB-, and O-.

Blood types are inherited: each biological parent donates one ABO allele (randomly chosen from their two) and one Rh factor allele to their child. Therefore 2 ABO alleles and 2 Rh factor alleles of the parents determine the child's blood type. For example, if both parents of a child have blood type A-, then the child could have either type A- or type O- blood. A child of parents with blood types A+ and B+ could have any blood type.

If the ABO allele pairs, represented as **strings**, are mapped to their respective blood types (using a **Map<string>**), and blood types (also expressed as **strings**) are reversed-mapped back to a **vector** of all possible ABO allele pairs (using a **Map<Vector<string> >**), then given the blood types of both mother and father, it's possible to compute the spectrum of possible blood types for their children. Likewise, given the blood type of one parent and a child of that parent, it's possible to infer the possible blood types of the second parent.

Use the next few pages to write two functions that infer possible blood types. The first will be written to generate the set of possible blood types a child could inherit from two biological parents. The second will be written using the first to infer a second parent's blood type from the first parent and the child. Go ahead and rip this page out, since you may want to read and reread it without flipping back and forth.

```
// utility function to generate properly ordered pairs from standalone alleles
string CreateAllele(char father, char mother)
{
    string allele; // empty C++ string to start, so that + just works
    if (father < mother)
        return allele + father + mother;
    return allele + mother + father;
}
```

- a. Present your implementation of the **GenerateInheritableBloodTypes** function that, given two blood types and the two maps described above, returns a string set containing all of the blood types a child could inherit from his or her parents. You should assume that the two maps are populated with data for the six associations above, and you may use the **CreateAllele** utility function to make sure the two alleles are properly ordered in the allele pair (so you generate **"AB"** instead of **"BA"**).

```
/**
 * Function: GenerateInheritableBloodTypes
 * Usage: types = GenerateInheritableBloodTypes("A-", "O+", alleles, bloodTypes);
 * -----
 * Given the two very small maps, generate the set of blood types that
 * a mother and father's biological child could inherit, given the mother's
 * and father's blood types. You may assume that all of the data is legitimate
 * so that you needn't do any error checking whatsoever. The two maps
 * only include allele and ABO blood types, but nothing about Rh factors.
 * So, alleleMap will include pairs like ("BO", "B"), and bloodTypeMap
 * will contain pairs like ("B", ["BB, BO"]) and ("O", ["OO"]).
 *
 * The set that's returned will include Rh factors with the blood types--
 * strings like "AB+", "AB-", and "O-" might be included.
 */
```

```

Set<string> GenerateInheritableBloodTypes(string mother, string father,
                                         Map<string>& alleleMap,
                                         Map<Vector<string> >& bloodTypeMap)
{

```

- b. (7 points) Now write the **InferPossibleParentBloodTypes** function, which computes and returns the set of possible blood types that could have combined with the known parent blood type to conceive a child with the child blood type. Do so by iterating over the key set of the **bloodTypeMap**, pairing each key up with a "+" or a "-", and tracking whether each blood type, when matched against the other parent blood type, could combine to produce the child blood type. You'll want to use your **GenerateInheritableBloodTypes** function from part a.

```

/**
 * Function: InferPossibleParentBloodTypes
 * Usage: possibilities =
 *         InferPossibleParentBloodTypes ("O-", "A+", alleles, bloodTypes);
 * -----
 * Information about a father's blood type and his child's blood type
 * places constraints on what the mother's blood type might be.
 * InferPossibleParentBloodTypes determines what the mother's blood
 * type might be by computing and returning the set of all possibilities.
 * This is done by iterating over all the keys of the bloodType map
 * to generate all blood-type/rh-factor pairs, and including those that
 * in principle could combine with the identified parent's blood type to
 * generate the child blood type.
 */

Set<string> InferPossibleParentBloodTypes(string parent, string child,
                                         Map<string>& alleleMap,
                                         Map<Vector<string> >& bloodTypeMap)
{

```