Assignment 9

Spell Checking Using Hash Tables

`README` Due:	11:00 PM,	Tuesday, November 25, 2003
Runnable Code #1 Due:	5:00 PM,	Wednesday, November 26, 2003
Runnable Code #2 Due:	11:00 PM,	Wednesday, December 3, 2003
Final Code Due:	11:00 PM,	Thursday, December 4, 2003

You are forbidden to work on this assignment over Thanksgiving break!

1 Scenario

One of the most widely used spelling checkers in Unix-like environments is `ispell`, written by our own Professor Kuenning.[1] Is Professor Kuenning's fame and glory deserved? Just how hard *is* it to write a spelling checker? You have decided to find out!

Like `ispell`, your spelling checker will use hash tables, but unlike `ispell`, your program will be written in C++ (`ispell` is written in C). To show your prowess at writing reusable C++ code, you have decided to make your program use a templated hash-table class.

2 High-Level Interface and Behavior

Your spelling checker, which is to be called `myspell`, will read text from standard input (`cin`) and report any words not present in a dictionary file, whose name is supplied on the command line.

The program begins by reading the dictionary into an internal hash table. It then reads words from standard input. Each word is looked up in the dictionary—words not in the dictionary are classified as being misspelled. Incorrectly spelled words are written to standard output in the order they are encountered, together with a list of suggested corrections.[2] (The algorithm for generating corrections is given in Section 4.)

Each line in the correction output should consist of the incorrect word followed by a colon (`:`) and zero or more corrections separated by spaces. There should be exactly one space after the colon (unless there are no corrections), and there should be no whitespace at the end of the line. The following example shows valid output:

```
unix% ./myspell smalldict.words < error-filled-file.txt
xyzzy:
foo: for
wird: bird gird ward word wild wind wire wiry
```

1. Professor Kuenning's `ispell` implementation is based on the first `ispell` implementation, written by Pace Willisson, and is more properly referred to as *International Ispell*.

2. This output format is similar to `ispell`'s `-a` mode.

Note: Your option processing code will not have to deal with (and will never see) the I/O redirection part of the command line (i.e., `< error-filled-file.txt`)—all Unix shells handle I/O redirection for you automatically, before your program is executed.

If a word is misspelled multiple times in the input file, it should only be printed once. If every word in the input is found in the dictionary, `myspell` should produce no output on cout.

For the purposes of spell checking, a *word* is a sequence of one or more characters for which isalpha is true. In the input text, words are separated by non-word characters. Although words in the input text may be mixed-case, your program will always convert words to lowercase (using tolower) before comparing them against words in the dictionary. The dictionary file is a sequence of unique lowercase words separated by whitespace.

Your myspell program must also support a debugging option, whereby the command `./myspell -d ispell.words` reads the dictionary `ispell.words` and performs spell checking as usual, but also prints information about the data structure used to represent the dictionary. This information should be printed immediately after the dictionary has been read, before spell checking begins, and sent to cerr. This output is required to be in the following format:

n `expansions, load factor` *f*, *c* `collisions, longest run` *l*

where *n*, *c*, and *l* are integers, and *f* is a floating-point (*double*) number, printed in the default format. This information corresponds to (or is easily calculated from) statistical information provided by the *HashSet* template class described in the next section. For example, `longest run` refers to the value returned by maximal.

If the program is given invalid arguments, it should produce appropriate error messages and quit gracefully rather than crash. The choice of error message is up to you.

3 Required Classes and Functions

In undertaking this assignment you will write a generic hash table class and a hash function for strings. These components must conform to the specifications given below.

3.1 Required *HashSet* Template Class

Your code must provide a template class *HashSet* (declared in `hashset.hpp`) such that *HashSet<string>* can store a hash table of *string*s. The *HashSet* class can assume that if a programmer creates an instance of the template *HashSet<Food>* (using *Food* as an example) they will have written a global function *size_t* myhash(***const*** *Food&* food)

that provides hash values for *Food*s using the full range of the size_t type (*size_t* is a synonym for *unsigned int*, defined by #include <cstddef>, with values ranging from 0 to $2^{32}-1$ on 32-bit machines).

An instance of *HashSet*, on some type *T* (i.e., *HashSet<T>*), will provide the following operations:

- A default constructor
- A destructor that leaves the object in a sane (empty) state
- A member function *size_t* size() ***const*** that returns the number of items stored in the hash table
- A member function *void* insert(***const*** *T&*), where h.insert(x) adds x to the hash table, where the function's behavior is undefined if x has already been added to the table
- A member function *bool* exists(***const*** *T&*) ***const***, where h.exists(x) returns true if x is present in the hash table and false otherwise
- A member function *size_t* buckets() ***const*** that returns the number of buckets in the hash table
- A member function *size_t* reallocations() ***const*** that returns the number of times the hash table has resized itself
- A member function *size_t* collisions() ***const*** that the number of times an insert into the current hash table representation has found a non-empty bucket[3]
- A member function *size_t* maximal() ***const*** that returns the length of the longest chain (in the case of separate chaining) or cluster (in the case of open addressing) discovered so far in the current hash table representation[4]

(The type *T* is used for illustrative purposes—you may not use such a short name for the template type argument in your template class.)

All of the above functions must execute in $O(1)$ expected amortized time.

Note that the last four functions provide statistics about the hash table—buckets, collisions, and maximal will change whenever the hash table needs to be expanded. In particular, collisions and maximal are statistics relating to the *current* hash table representation—when a hash table is resized, throw away any data you were storing for these values and start over.

You are *not* required to provide a copy constructor, assignment operator, a function to print the hash table, a swap function, or an erase function. You are not required

3. If you use an open addressing representation, collisions should only care about the first non-empty bucket.

4. The "discovered so far" requirement is there because it may not be easy to know the actual largest cluster size without scanning the entire table. You may wish to think about why.

to write an iterator for this class. If copy construction and assignment are not supported, they must be disabled.

Your hash table will need to grow as necessary to hold its contents while keeping the load factor within a range that offers good performance. It is up to you to decide how to handle collisions—separate chaining, linear probing, quadratic probing, or double hashing. If `maximal` returns a value larger than about 20, it is a sign of serious problems with your hash table, either because your program is not growing the hash table appropriately, or because of serious problems with your hash function.

Choosing a good load factor is up to you. Part of the choice will be determined by whether your hash table uses separate chaining or open addressing techniques. Whichever representation you use, it is sensible to define a constant that will influence or control the load factor of your hash table, and then experiment with different values. You may want to use performance measures to determine an appropriate size—in earlier assignments you learned how to conduct simple benchmark tests using the `time` command—but remember that programs require space as well as time. You should include a description of how you arrived at your load factor in your `README` file.

Your implementation may use any data type from the STL as well as the *slist* type, which is an extension to the standard STL present in our GNU STL library.[5] For obvious reasons, you may not use the *hash_set* or *hash_map* SGI extensions present in our STL library, except (perhaps) for testing.

3.2 Required `myhash` Function

You must also provide a function to hash strings, with the following signature:

```
size_t myhash(const string& s)
```

It must be declared in the header `stringhash.hpp` and defined in the file `stringhash.cpp`. The header file for your *HashSet* template class should *not* automatically include this header file.

You are not expected to devise your own hashing strategy. You may copy code from *any* source to make your hash function, with the exception of the work of prior CS 70 students and complete hash-table implementations. You may find appropriate functions in Weiss. Your hash function should be a good one, however. If you do a web search for hash functions, you should be careful not to copy implementation code for hash tables themselves—you only have dispensation to find interesting hash functions.

You should also test your code with a *very bad* hash function. We will. (Your program only has to meet its complexity obligations for a good hash function. It should, however, behave sanely when given a bad hash function.)

5. The *slist* template class is defined by the line `#include <ext/slist>` and resides in the `__gnu_cxx` namespace, not the `std` namespace.

4 Generating Spelling Corrections

The easiest way to generate corrections in a spelling checker is by trial and error. If we assume that the misspelled word contains only a single error, we can try all possible corrections and look each up in the dictionary.

Traditionally, spelling checkers have looked for four possible errors: a wrong letter ("`wird`"), an inserted letter ("`woprd`"), a omitted letter ("`wrd`"), or a pair of adjacent transposed letters ("`wrod`"). To simplify this assignment, you will only need to deal with the first possibility: a wrong letter. When a word isn't found in the dictionary, you will need to look up all variants that can be generated by changing one letter. For example, given "`wird`", you should look up "`aird`", "`bird`", "`cird`", ..., "`zird`", then "`ward`", "`wbrd`", "`wcrd`", ..., "`wzrd`", and so forth. Whenever you find a match in the dictionary, you should add it to your output line.

5 Sample Dictionaries

You may wish to create a very small sample dictionary of your own for initial testing. A slightly larger dictionary of 341 words (`simple-dict.words`) should help you to get most of your bugs out. When you're fairly confident, you can try your luck with over 34,000 words in an all-lowercase version of the `ispell` dictionary, `ispell.words`. Both dictionaries are available for download from the Homework area of the course website. You do not need to include these dictionaries in your submission.

6 Tricky Stuff

As usual, there are parts of this assignment that are tricky. Here are a few hints and tips:

6.1 Template Tips

- Templates aren't compiled by themselves. There will be no `hashset.o` file, and so there is no `hashset.cpp`—instead the implementation code goes into a private header file that that is #included at the end of the interface header.
- Remember that all the things your template needs (e.g., the iostream library) should be #included before the template code is read by the compiler. Confusingly, the compiler *won't* always complain when it sees the template code—it won't realize that there is a problem until you instantiate the template for a particular type.
- You can use an explicit instantiation of your entire template to check for syntax errors that wouldn't be caught otherwise. One option is to write a `.cpp` file for use in your own personal testing containing the following code:

```
// Force C++ to perform some sanity checking on my template code.
#include "hashset.hpp"

size_t hash(int);
template class HashSet<int>;        // explicit instantiation of entire templated class
```

(Notice that the code uses *int*s and not *string*s—that way you can be sure you haven't introduced any subtle dependencies on *string*s in code that is supposed to be generic.)

- There are some subtleties about using nested classes with templates, as well as template classes that use template classes. For example, suppose you had written the following templated top-level function to print out any list:

```
#include <iostream>
#include <list>
#include <string>
using namespace std;

template <typename Element>
void printAnyList(list<Element>& things)
{
    for (list<Element>::iterator i = things.begin(); i != things.end(); ++i)
        cout << *i << endl;
}

int main(int argc, const char* argv[])
{
    list<string> args(argv, argv+argc);
    printAnyList(args);
    return 0;
}
```

This code should look clear and correct—to a human. But if you compile this code, you could be faced with the following weird error:

```
example.cpp: In function ‘void printAnyList(std::list<Element>&)’:
example.cpp:9: parse error before ‘=’ token
example.cpp: In function ‘void printAnyList(std::list<Element>&)
    [with Element = std::string]’:
example.cpp:16:   instantiated from here
example.cpp:9: ‘i’ undeclared (first use this function)
example.cpp:9: (Each undeclared identifier is reported only once for each
    function it appears in.)
```

The compiler isn't recognizing that *list<Element>::iterator* is the name of a type and thus sees our code as being syntactically incorrect.

This error probably seems weird—why doesn't the compiler understand what we've written? The cause is subtle. When the compiler parses the template definition, it can't be sure what *list<Element>::iterator* really is, because it doesn't know what *Element* is yet—for all it knows, iterator could be the name of some public classwide constant or variable (after all, to the compiler, there is nothing special about the word iterator, it is just an identifier, little different from the identifier MAX_SIZE—we know to expect one to be a type and the other to be a constant, but the compiler can't make that distinction without more information). In other words, because the compiler can't instantiate *list<Element>* until it knows what *Element* is, the compiler can't know *anything* about what the *list<Element>* class is like, so it has to *guess* what *list<Element>::*iterator is. When a C++ compiler examines template code and doesn't know whether an identifier is a type or not, the compiler assumes that it *is not* a type. Even though you later instantiate the template using *string* as the *Element* type, and even though the compiler will then know that *list<string>::iterator* is a type, it doesn't matter. The code has already been parsed and it is *too late*.

What you actually need to do is let the compiler know that you expect is a type by prefixing the offending type with the keyword **typename**, as follows:

```
for (typename list<Element>::iterator i = things.begin(); i != things.end(); ++i)
    cout << *i << endl;
```

This code works, but looks pretty ugly and longwinded. Remember, though, if you're doing this kind of thing with a class, then your class can have a private **typedef** to avoid typing this kind of thing over and over.

In recent versions of g++, they have made the compiler behave more helpfully for this kind of error. Now g++ gives a warning similar to the one shown below for this issue.

```
example.cpp: In function 'void printAnyList(std::list<Element>&)':
example.cpp:9: warning: 'std::list<Element>::iterator'
   is implicitly a typename
example.cpp:9: warning: implicit typename is deprecated, please see the
   documentation for details
```

- Template error messages can be pretty confusing. If you have a hard time figuring out what the compiler is telling you, don't spend hours scratching your head, ask for help.

6.2 Hash Table Help

- If you define a constant for your desired load factor, remember that only constants of integral type (e.g., *int*s) can be declared and defined at the same time inside the class declaration. Constants of other types (e.g., *double*s) must be declared in the class declaration and defined in the implementation file.

- When you are developing a hash table, it is wise to start your debugging with a dummy hash function that always returns zero. Once you are sure your collision handling works correctly, you can write a real hash function.
- If you get excessive numbers of collisions, be sure your hash function is returning values that are well spread out. Test it with "nearby" values such as `aaa`, `aab`, `baa`, and so forth.
- Remember that the hash functions can return a number larger than the table size. You must reduce the hash value yourself to make sure you don't go beyond your array bounds.
- Remember that when you expand your hash table, you must rehash everything in the current table, because the wrapping due to the modulus operator will change. If you are using separate chaining, don't forget to follow your collision chains appropriately.
- Similarly, you may be tempted to use the *vector* class from the library to manage your hash buckets. Although it is possible to do so effectively, doing so is trickier than it might first appear. In particular, the `resize` function is *not* appropriate for resizing hash tables. Many students find it easier to avoid using *vector*, and simply manage the array of hash buckets themselves. If you do use *vector*, the most sensible approach is to build the new hash table in a new vector and then `swap` it.