CS116 - Intro to Programming, C++
Summer 1997

Project 2

Spell Checker - Part 1

Introduction
In this project you will implement a Macintosh based spell checker, much like the ones used in commercial wordprocessors. You will start by implementing a simple dictionary class and using it to build a text based spell checker. After that, you will extend the dictionary to implement sorting and fast searching, and will use it to write a graphical spell checker.

Problem 1. File and String Warm-up
[ 10 pts: cSeisit3 ]

Write a program that reads a file of words, with one per line, and outputs two things:

• The longest word in the file,
• The number of words of each length in the file.

In your program, declare a const int MaxWordLength set to 25, which will represent the maximum length of any word you might encounter. Use this constant appropriately in your program.

There are two files for you to test your program on. They are

• shorttest.text, which contains one word of each length up to length 10,
• longtest.text, which contains a much longer list of words.

Both files are on the Maclab server in the CS 116 : Project 2 folder. You should copy these files into the same directory as your project for easiest use.
Problem 2. Building a Dictionary

The goal of this problem is to create a generic "dictionary" class. As part of its constructor, a dictionary object will read in a set of words from a file. You will then be able to ask the dictionary whether it contains a given word or not.

Part I

[ 20 pts: c10e2s3t5 ]

Although we could let the words in our dictionary be simply strings (i.e. \0-terminated arrays of characters), we will use a more abstract (and modular) approach and design a class for words. Design a class CWord with the following members:

- A member variable string which represents the actual string associated with the object. This means that string will simply be a \0-terminated array of chars.
- A constructor with no arguments which sets string to the empty string (the empty string is an array of one char - which one?)
- Another constructor which takes a string as argument and initializes the private member string appropriately. Note that for this a simple assignment of pointers will not suffice; you must use the library function strcpy in some way (Why?).
- A destructor which does the necessary house cleaning (if any).
- Two members getString and setString which do the appropriate things. Think carefully about memory allocation and deallocation here.
- Operators ==, <= and >= appropriately overloaded to implement word comparisons. The return type of these operators should be bool, and comparisons should be done lexicographically (i.e. as in phonebooks), as done by strcmp.

You should break the code for class CWord into a header file CWord.h and a source file CWord.cpp as discussed in class.

Test your class extensively to make sure it works. You can start your testing with something like this:

CWord word1, word2;
CWord word3(‘hello’);
word1 == word2; // this should return true
word1 == word3; // this should return false
word1.setString(‘hello’);
word1 == word3; // this should return true
Part II

[ 20 pts: c10e2s3t5 ]

Design a class **CDictionary** with the following members:

- A private array **words** of **CWords** which will hold the words in the dictionary object. This array should be dynamically allocated (Why?).

- A constructor which takes a file name (i.e. string) as an argument and initializes the **words** array to hold all the words in the file. Note that you must first allocate the right amount of memory for **words** and then initialize the elements of the array. (How many times will you have to read the file? )

- A destructor which takes care of the necessary memory deallocation.

- A member function **wordInDictionary** which takes an argument of type **CWord** and returns **true** or **false** appropriately.

Again you should split your code into a header file **CDictionary.h** and a sourcefile **CDictionary.cpp**.

Fully test your **CDictionary** class to make certain that it works properly. Here is a sample test:

```cpp
CDictionary myDictionary(‘shortest.text’);
CDictionary yourDictionary; // should be illegal
CWord myWord(‘one’);
CWord yourWord(‘yo!’);
myDictionary.wordInDictionary(myWord); // should return true
myDictionary.wordInDictionary(yourWord); // should return false
```
Part III

[ 20 pts: c10e2s3t5 ]

Now you will integrate your two classes CWord and CDictionary into a simple spell checker. Write a function spellCheckFile which takes an input file name and an output file name and a CDictionary* as its three arguments. The function should read the input file and write a modified copy of the input file to the output file. The modifications are as follows:

- Incorrect words should be marked with delimiters. For example, ther in the input file should be replaced by !ther! in the output file (assuming of course that there is no word with string ther in the dictionary).
- Lines which have spelling errors in them should be marked at the end. For example, if the second line of the input file is

  everything lazy, everthing hazy

  the second line of the output file should be

  everything lazy, !everthing! hazy *** problem at line 2 ***

If you want you can set a reasonable limit on the size of the input lines, say 100 chars or so (Needless to say, this should be done with a const int MaxInputLineLength initialized to the limit). Test your spell checker on the file spelling.text. As your dictionary you should use the file dictionary.text.

Note. One pitfall in writing your spell checker is trying to do everything in one function. This is bad style and should be avoided. As a general rule, if a function is more than 15 or 20 lines long, it is too long and parts of it should be abstracted out and made into separate functions which are then called from the original function. In order to do this, you may end up passing a few arguments into these “helper” functions. This is a common thing to do. As an example, you may want to write a function spellCheckLine, which takes a line of text and a CDictionary* as arguments, modifies the line of text to include the delimiters around the misspelled words, and returns true or false depending on whether the line had any spelling errors in it. (What would be the use of this return value to the calling function?)