

CS116 - Intro to Programming, C++

Summer 1998

Project 2

Spell Checker

1 Introduction

In this project you will implement a spell checker, much like the ones used in commercial word processors. You will start by implementing a simple dictionary class and using it to build a simple spell checker. After that, you will extend the dictionary to implement sorting and fast searching and will use it to write a menu driven spell checker.

Note: In this project, we will not use the C++ `string` class, but instead will only use `'\0'`-terminated character arrays. So from now on a *string* refers to such an array and not the class.

To get started, use the following command to copy a few files into your own directory:

```
cp -pr ~behfar/code/project2 project2
```

2 File and String Warm-up

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

- The longest string in the file,
- The number of strings of each length in the file.

In your program, have a constant set to **25**, which will represent the maximum length of any string you might encounter. Use this constant appropriately in your program.

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

- `shorttest`, which contains one string of each length up to length 10.
- `longtest`, which contains a much longer list of strings.

3 Building a Dictionary

We start work on our spell checker by creating a generic “dictionary” class. As part of its constructor, a dictionary object will read in a set of strings from a file. We will then be able to ask the dictionary whether it contains a given word or not.

The Word Class

Although we could let the words in our dictionary be simply strings (i.e. character arrays), we will try a more abstract (and modular) approach and use the `Word` class discussed in class. This will allow us to work with the underlying strings in a more intuitive way.

An object of class `Word` has a (private) member variable `string` which represents the actual string being stored. A constructor with no arguments, a constructor which takes a string as argument, and a destructor should be provided. Two member functions for reading and writing the private string are also needed. Note that all the above functions must carefully work together to ensure proper memory allocation and deallocation of character arrays.

In addition, the operators `==`, `<=` and `>=` should be appropriately overloaded to implement lexicographic word comparisons (i.e. as in phonebooks).

As usual, you should break the code for class `Word` into a header file `Word.h` and a source file `Word.cc`, and provide the proper include guards for the header.

Test your class extensively. You can start with something like this:

```
Word word1, word2;
Word word3("hello");
word1 == word2; // this should return true
word1 == word3; // this should return false
word1.setString("hello");
word1 == word3; // this should return true
```

The Dictionary Class

A `Dictionary` will have a private array of `Words` which will hold the words in the dictionary. A constructor should be provided which takes a file name (i.e. string) as an argument and initializes the array of words to hold the strings in the file (Don’t forget to provide an appropriate destructor). A member function `wordInDictionary` for checking whether a given word is in the dictionary should also be provided.

As with the `Word` class you should split your code into a header file `Dictionary.h` and a source file `CDictionary.cc`, and provide include guards.

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

```

Dictionary myDictionary("shorttest");
Dictionary yourDictionary; // illegal: no default constructor
Word myWord("one");
Word yourWord("yo!");
myDictionary.wordInDictionary(myWord); // should return true
myDictionary.wordInDictionary(yourWord); // should return false

```

A Simple Spell Checker

Now you will integrate your two classes `Word` and `Dictionary` into a simple spell checker. Write a function `spellCheckFile` which takes an input file name and an output file name and a `Dictionary*` 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, everything up in the air
```

then the second line of the output file should be

```
everything lazy, !everthing! hazy, everything up in the air *** problem at line 2 ***
```

If you want you can set a reasonable limit on the size of the input lines, say 100 characters or so (Needless to say, this should be done with a `const...`). Test your spell checker on the file `spelltest`. As your dictionary you should use the file `dict`.

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 something indicating whether the line had any spelling errors in it. (What would be the use of this return value to the calling function?)

4 Extending The Dictionary

Chances are that your implementation of the search function `wordInDictionary` uses *linear search*. This approach is inefficient, and in this problem you will extend your `Dictionary` class to implement the faster *binary search*. You will also enable your dictionary to learn, i.e. you will be able to add words to it.

Sorting and Searching

Linear search is a particularly slow search method and will not do for a practical spell checker. To make your dictionary efficient you need to implement a much faster search method.

Sorting. Change the `Dictionary` class to have an array of `Word pointers` rather than an array of `Words`. Write a member function `sort` for your `Dictionary` class that sorts the array of words using the quicksort algorithm discussed in class. You must implement quicksort yourself rather than use a library function. (Important question: does `sort` need to be part of the public interface?) The fact that your dictionary has an array of `Word pointers` makes swapping elements more efficient and hence improves the performance of `sort`. Test your `sort` routine fully to make sure it works properly. Use the unsorted dictionary file `dict` provided for you. You may want to add a member function `displaySelf` to make testing easier.

Searching. Now you can modify the member function `wordInDictionary` to use binary search on the (sorted) array of your dictionary. Test this function to make sure it works properly, both on words that are in the dictionary and on ones that are not (you may use the *sorted* dictionary file `sorteddict` for testing purposes in case you want to work on this problem before you implement your `sort` routine).

In a typical spell checker, one is able to add words to the dictionary. Your spell checker application will provide this feature by having two dictionaries, a main dictionary and a personal dictionary. The main dictionary gets its words from some standard file (like `dict`) and we do not plan to add any words to it. The personal dictionary however should allow for the addition of new words, as follows.

Adding Words. Add a new constructor to the `Dictionary` class that takes an integer argument representing the maximum size of the dictionary. It should allocate an array of the given size (but record the fact that there are no words in the dictionary yet). Add another constructor that takes a file name *and* an integer size and creates a dictionary of that size, reading into it as many words as are provided by the file. Add a member `addWord` which takes a word as argument and adds the word to the dictionary. Add another member `addEntry` which takes a string and adds the corresponding word to the dictionary. Finally,

add a member `save` that takes a file name as argument and saves the contents of the dictionary to that file so that it can be read again using your constructor. Again a member function `displaySelf` will come in handy for testing your functions.

The Final Spell Checker

Now you are ready to implement the spell checker application. Your program should be menu driven. The main menu should provide for reading in the main dictionary (if one is not loaded already), reading in a personal dictionary, saving the personal dictionary to a file, spell checking a file, and writing the spell checked result to an output file. When spell checking a file, upon encountering a misspelled word, your program should display the misspelled word, and prompt the user for one of the following options:

- **Skip.** The word is treated as if it were spelled correctly.
- **Mark.** The word is marked as incorrect (as in the simple spell checker).
- **Add.** The word is added to the personal dictionary.
- **Correct.** The user is prompted for a correction.

Test your application extensively. You should make up your own text files which contain spelling errors and test your application on those.

5 Extra Credit

Here are a two suggestions for improving your program:

- Make the word comparisons case insensitive, proceeding as follows. Add a member function `isEqual` to your `CWord` class which takes a word and an optional parameter `case` of type `ComparisonType` as arguments. `ComparisonType` should be an enumerated type with two possible values: `CaseSensitive` and `CaseInsensitive`. `case` should default to `CaseInsensitive`, and it should indicate how `isEqual` is to perform the comparison. Now use `isEqual` instead of the `==` operator in your dictionary's sort and search routines.
- Remove the size restriction of the personal dictionary, so the user can add as many words to it as desired. This will involve deallocation of the old array of words and allocation of a bigger array of words, but it should be completely transparent to the user.

There are many other ways in which the spell checker could be extended. Feel free to talk to me about any specific things you have in mind, or ask me for more suggestions.