MSc (Computing Science) 2011-2012
C/C++ Laboratory Examination

Imperial College London

Monday 9 January 2012, 15h00 – 17h15

> "A human being should be able to change a diaper, plan an invasion, butcher a hog, conn a ship, design a building, **identify a sonnet**, balance accounts, build a wall, take orders, give orders, cooperate, act alone, solve equations, **analyze a new problem**, pitch manure, **program a computer**, cook a tasty meal, fight efficiently, die gallantly."
>
> *Robert Heinlein (mostly)*

☞ You must complete and submit a working program by 17h15.

☞ Log into the Lexis exam system using your DoC login as both your login and as your password (**do not use your usual password**).

☞ You are required to add to the pre-supplied header file **sonnet.h**, pre-supplied implementation file **sonnet.cpp** and to create a **makefile** according to the specifications overleaf.

☞ You will find the source files **sonnet.cpp**, **sonnet.h** and **main.cpp**, and the data files **shakespeare.txt**, **petrarch.txt**, **spenser.txt**, **mystery.txt** and **dictionary.txt** in your Lexis home directory (**/exam**). If you are missing one of these files please alert one of the invigilators.

☞ **Save your work regularly.**

☞ Please log out once the exam has finished. No further action needs to be taken to submit your files – the final state of your Lexis home directory (**/exam**) will be your submission.

☞ No communication with any other student or with any other computer is permitted.

☞ You are not allowed to leave the lab during the first 30 minutes or the last 30 minutes.

☞ **This question paper consists of 6 pages.**

Figure 1: Famous sonnet writers Shakespeare (left), Petrarch (middle) and Spenser (right).

Problem Description

You are challenged to write a computer program that can identify three styles of sonnet. A sonnet[1] is a poem of fourteen lines that follows a *rhyme scheme* according to its *style*. A rhyme scheme is a sequence of alphabetic letters (always beginning with *a*) which reflects the pattern of rhymes occurring at the end of the lines in a poem.

The figure above shows three of the most prolific writers of sonnets. Each of them developed their own style of sonnet. A *Shakespearean* sonnet has rhyme scheme *ababcdcdefefgg* as shown below:

Shall I compare thee to a Summer's **day**? *a*
Thou art more lovely and more **temperate**: *b*
Rough winds do shake the darling buds of **May**, *a*
And Summer's lease hath all too short a **date**: *b*
Sometime too hot the eye of heaven **shines**, *c*
And oft' is his gold complexion **dimmed**; *d*
And every fair from fair sometime **declines**, *c*
By chance or nature's changing course **untrimmed**; *d*
But thy eternal Summer shall not **fade** *e*
Nor lose possession of that fair thou **owest**; *f*
Nor shall Death brag thou wanderest in his **shade**, *e*
When in eternal lines to time thou **growest**: *f*
So long as men can breathe, or eyes can **see**, *g*
So long lives this, and this gives life to **thee**. *g*

Likewise, a *Petrarchan* sonnet has rhyme scheme *abbaabbacdcdcd* and a *Spenserian* sonnet has rhyme scheme *ababbcbccdcdee*.

[1] The term sonnet derives from the Italian word *sonnetto* meaning "little song".

Pre-supplied functions and files

To get you started, you are supplied with some helper functions (with prototypes in **sonnet.h** and implementations in the file **sonnet.cpp**):

1. `bool get_word(const char *input_line, int word_number, char *output_word)` is a helper function that can be used to retrieve (via the output parameter `output_word`) a word (specified by its `word_number`) in a line of text (given by `input_line`). If the `word_number` is invalid the function returns false and the `output_word` is empty; otherwise the function returns true and the `output_word` is in uppercase.

   For example, the code:

   ```
   char word[512];
   bool success = get_word("One, two, three!", 2, word);
   ```

   results in `word` set to `"TWO"`, and `success` set to `true`.

2. `char rhyming_letter(const char *ending)` will generate the rhyme scheme letter (starting with *a*) that corresponds to a given line `ending`. The function remembers its state between calls using an internal lookup table, such that subsequents calls with different `endings` will generate new letters. The state can be reset (e.g. to start issuing rhyme scheme letters for a new poem) by calling `rhyming_letter(RESET)`.

   For example, the code:

   ```
   char one, two, three, four;
   rhyming_letter(RESET);
   one = rhyming_letter("AY");
   two = rhyming_letter("ATE");
   three = rhyming_letter("AY");
   four = rhyming_letter("ATE");
   ```

   results in `one`, `two`, `three` and `four` set to `'a'`, `'b'`, `'a'` and `'b'` respectively.

You are also supplied with a main program in **main.cpp** and four example sonnets in **shakespeare.txt**, **petrarch.txt**, **spenser.txt** and **mystery.txt**.

Finally, but importantly in the context of rhyme detection, you are also given a phonetic dictionary **dictionary.txt**[2]. This shows how words can be broken up into fundamental sound units called phonemes. Each entry consists of a word followed by its phonemes, e.g.:

```
DAY        D EY
MAY        M EY
CONVICT    K AA N V IH K T
PICKED     P IH K T
```

To decide whether two words rhyme we construct and compare their *phonetic endings*. If these match, we conclude that the words rhyme; otherwise, we conclude that they do not. The phonetic ending of a word is constructed by concatenating the *last phoneme of the word which contains a vowel*[3] with all subsequent phonemes of the word (if any). For example, the phonetic ending of both `DAY` and `MAY` is `EY`, and the phonetic ending of both `CONVICT` and `PICKED` is `IHKT`.

Specific Tasks

1. Write a function `count_words(line)` which returns the number of words in a given input string `line`. For example, the code:

   ```
   int words = count_words("It’s not so easy!");
   ```

   results in `words` having the value 4.

2. Write a function `find_phonetic_ending(word, phonetic_ending)` which uses the phonetic dictionary in the file **dictionary.txt** to construct the phonetic ending for the (uppercase) word contained in the input parameter `word`. If this word is in the phonetic dictionary, the corresponding phonetic ending should be stored in the output parameter `phonetic_ending`, and the function should return `true`. Otherwise the function should return `false`.

   For example, the code:

   ```
   char ending[512];
   bool success = find_phonetic_ending("CONVICT", ending);
   ```

   results in `ending` set to `"IHKT"` and `success` set to `true`.

[2] A simplified version of the CMU Pronouncing Dictionary (Credit: Carnegie Mellon University).

[3] That is, one of `a`, `e`, `i`, `o` or `u`.

3. Write a function `find_rhyme_scheme(filename, scheme)` which produces in the output parameter `scheme` the rhyme scheme for the sonnet contained in the file `filename`. If the file does not exist, the function should return false; otherwise the function should return true.

   For example, presuming the file **shakespeare.txt** contains the Shakespearean sonnet shown in the problem description, the code:

   ```
   char scheme[512];
   bool success;
   success = find_rhyme_scheme("shakespeare.txt", scheme);
   ```

   results in `scheme` set to `"ababcdcdefefgg"` and `success` set to `true`.

4. Write a function `identify_sonnet(filename)` which has as its return value one of the strings: `"Shakespearean"`, `"Petrarchan"`, `"Spenserian"`, or `"Unknown"` according to whether the rhyme scheme of the sonnet in file `filename` matches that of a Shakespearean, Petrarchan or Spenserian sonnet; if no match can be found then the string `"Unknown"` should be returned.

   For example, if the file **spenser.txt** contains a Spenserian sonnet, the code:

   ```
   cout << "The sonnet spenser.txt is a " <<
     identify_sonnet("spenser.txt") << " sonnet" << endl;
   ```

   should generate the output:

   ```
   The sonnet spenser.txt is a Spenserian sonnet
   ```

Place your function implementations in the file **sonnet.cpp** and corresponding function declarations in the file **sonnet.h**. Use the file **main.cpp** to test your functions. Create a **makefile** which compiles your submission into an executable file called **sonnet**.

(The four parts carry, respectively, 20%, 35%, 25% and 20% of the marks)

Bonus Challenge

For bonus credit *when you are fully satisfied with your other answers*, rewrite the `rhyming_letter(...)` helper function without using any STL classes (e.g. `string` and `map`). In doing so you may add additional helper functions as necessary.

Hints

1. You will save a lot of time if you begin by studying the main program in **main.cpp**, the pre-supplied functions in **sonnet.cpp**, the phonetic dictionary **dictionary.txt** and the sample sonnets **shakespeare.txt**, **petrarchan.txt**, **spenserian.txt** and **mystery.txt**.
2. Questions 1, 2 and 3 will be much easier if you exploit the pre-supplied functions.
3. To produce an initial (albeit incorrect and somewhat rough-and-ready) scaffold for Question 2 which catches most rhymes, simply set the output parameter to be a string made up of the last two letters of the last word in the input line.
4. Feel free to define any auxiliary functions which would help to make your code more elegant. For example, in Question 2, an auxiliary function which determines if a word includes a vowel may be useful.
5. Try to attempt all questions. If you cannot get one of the questions to work, try the next one.
6. You are not explicitly required to use recursion in your answers to any of the questions. Of course, however, you are free to make use of recursion if you wish (esp. where it increases the elegance of your solution).