MSc (Computing Science) 2018-2019
C/C++ Laboratory Examination

Imperial College London

Monday 7 January 2019, 14h00 – 16h00

DOUBLETS

A WORD-PUZZLE

BY

LEWIS CARROLL

"Double, double,
Toil and trouble."

London:

MACMILLAN AND CO

1879

- ☞ You are advised to use the first 10 minutes to read through the questions.
- ☞ 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 must add to the pre-supplied header file **doublets.h**, pre-supplied implementation file **doublets.cpp** and must create a **makefile** according to the specifications overleaf. You are also pre-supplied with a header file **dictionary.h** which you do not have to modify.
- ☞ You will find source files **doublets.cpp**, **doublets.h**, **dictionary.h**, and **main.cpp**, and data file **words.txt** in your Lexis home directory (**/exam**). If one of these files is missing alert 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.
- ☞ 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 15 minutes.
- ☞ **This question paper consists of 6 pages.**

Problem Description

W H E A T
c h e a t
c h e a p
c h e e p
c r e e p
c r e e d
b r e e d
B R E A D

Figure 1: Charles L. Dodgson a.k.a. Lewis Carroll (left) and an example of a Doublets chain (right)

Doublets (also known as Word Ladders, Word-Links, Laddergrams and Word Golf) is a word game invented in the 1870s by Charles L. Dodgson, (shown on the left in Figure 1) using the pen name Lewis Carroll[1].

The aim of Doublets is to form a *chain* of words which transforms a given *start word* into a given *target word* (which must be of the same length as the start word) in a number of steps. At each step, one (and only one) letter may be changed in the word from the previous step, provided that the new word so formed appears in an approved dictionary of English words (which Dodgson published as a *glossary*). Words that have already appeared in previous steps are not permitted. Figure 2 shows pertinent extracts from the original rules.

---

1. The words given to be linked together constitute a "Doublet;" the interposed words are the "Links;" and the entire series a "Chain."

2. Each word in the Chain must be formed from the preceding word by changing one letter in it, and one only. The substituted letter must occupy the same place, in the word so formed, which the discarded letter occupied in the preceding word, and all the other letters must retain their places.

3. When three or more words are given to be made into a Chain, the first and last constitute the "Doublet." The others are called "Set Links," and must be introduced into the Chain in the order in which they are given. A Chain of this kind must not contain any word twice over.

4. No word is admissible as a Link unless it (or, if it be an inflection, a word from which it comes) is to be found in the following Glossary.

Figure 2: Extracts of the rules of Doublets as published in a March 1879 edition of Vanity Fair magazine

An example of a Doublet chain in which the start word "WHEAT" is transformed into the target word "BREAD" in seven steps is shown on the right in Figure 1. **Note it is conventional when presenting a chain to display the start words and target words in uppercase while words in intermediate steps (i.e. the links) are displayed in lowercase.**

[1]Works published by Dodgson using the same pen name include *Alice in Wonderland* and the poem *Jabberwocky*.

Pre-supplied functions and files

You are supplied with a main program in **main.cpp**, and a data file **words.txt** containing a dictionary of valid English words[2] in uppercase. An extract of **words.txt** is presented below:

```
...
ADAGES
ADAPT
ADAPTABLE
ADAPTER
...
JUGS
JUMBLE
JUMBLES
JUMP
...
WHATS
WHEAT
WHEATEN
WHEATLESS
...
```

You are also supplied with the beginnings of the header file **doublets.h** (for your function prototypes) and the beginnings of the implementation file **doublets.cpp** (for your function definitions).

The file **doublets.cpp** includes the definition of a pre-supplied function:

```
bool dictionary_search(const char *word)
```

which is designed to return `true` if the given uppercase input string `word` is present in the file **words.txt**. For example, the code:

```
  bool found = dictionary_search("ADAPTABLE");
  cout << "The word 'ADAPTABLE' is " << (found ? "" : "NOT")
       << "found in the dictionary." << endl << endl;

  found = dictionary_search("JUMBLEWOCK");
  cout << "The word 'JUMBLEWOCK' is " << (found ? "" : "NOT ")
       << "found in the dictionary." << endl << endl;
```

has the output:

```
The word 'ADAPTABLE' is found in the dictionary.

The word 'JUMBLEWOCK' is NOT found in the dictionary.
```

Note that the `dictionary_search(...)` function makes use of a Dictionary class declared in **dictionary.h**. For the purposes of this exercise, you do not need to understand how the Dictionary class works, or make use of the Dictionary class yourself. If you are nevertheless curious as to how it works, feel free to look up Bloom filters after the examination[3].

[2] Painstakingly adapted from the original glossary published by Dodgson.

[3] See e.g. `https://en.wikipedia.org/wiki/Bloom_filter`.

Specific Tasks

1. Write a Boolean function `valid_step(current_word, next_word)` which returns `true` if the step from `current_word` to `next_word` represents a valid step in a Doublet chain according to Rule 2 and Rule 4 of Figure 2. You may assume that both words are supplied in uppercase format[4]. For example, the code:

   ```
   bool valid = valid_step("WHEAT", "CHEAT");
   cout << "From 'WHEAT' to 'CHEAT' is " << (valid ? "" : "NOT ")
        << "a valid step." << endl;
   ```

   should display the output

   ```
   From 'WHEAT' to 'CHEAT' is a valid step.
   ```

   Similarly, the code

   ```
   bool valid = valid_step("WHEAT", "WHEAD");
   cout << "From 'WHEAT' to 'WHEAD' is " << (valid ? "" : "NOT ")
        << "a valid step." << endl;
   ```

   should display the output

   ```
   From 'WHEAT' to 'WHEAD' is NOT a valid step.
   ```

   This is because, although the words differ only in their final letters (obeying Rule 2), 'WHEAD' does not appear in the dictionary of approved words (violating Rule 4).

2. Write a Boolean function `display_chain(chain, output_stream)` which writes a given chain to an output stream according to the conventions for the presentation of chains described in the Problem Description. The input parameter `chain` is a NULL-terminated array of uppercase C-style strings representing the words found at each step of the chain, while the input parameter `output_stream` can be any valid output stream, including `cout`. The function should return `true` if the entire chain was successfully written to the output stream, and `false` otherwise. For example, the code:

   ```
   const char *wheat_chain[] = { "WHEAT", "CHEAT", "CHEAP", "CHEEP",
     "CREEP", "CREED", "BREED", "BREAD", NULL };

   cout << "Displaying 7-step chain from 'WHEAT' to 'BREAD':" << endl;
   bool success = display_chain(wheat_chain, cout);
   cout << "Output " << (success ? "successful" : "failed") << "!" << endl;
   ```

   should display the output

   ```
   Displaying 7-step chain from 'WHEAT' to 'BREAD':
   WHEAT
   cheat
   cheap
   cheep
   creep
   creed
   breed
   BREAD
   Output successful!
   ```

[4] Throughout this exercise, you may assume that all input words will be uppercase. Lowercase letters will only be used when presenting the links of a chain as output.

3. Write a Boolean function `valid_chain(chain)` which returns `true` if and only if the given `chain` is a valid Doublets chain according to all four rules of Figure 2. As before, the input parameter `chain` is a NULL-terminated array of uppercase C-style strings representing the words found at each step of the chain. For example, the code:

```
const char *wheat_chain[] = { "WHEAT", "CHEAT", "CHEAP", "CHEEP",
  "CREEP", "CREED", "BREED", "BREAD", NULL };

bool valid = valid_chain(wheat_chain);
cout << "The chain from 'WHEAT' to 'BREAD' is " << (valid ? "" : "NOT ")
     << "a valid chain." << endl;
```

should result in the output

```
The chain from 'WHEAT' to 'BREAD' is a valid chain.
```

Likewise, the code:

```
const char *repeat_chain[] = { "WHEAT", "CHEAP", "WHEAT", "CHEAP", NULL };

bool valid = valid_chain(repeat_chain);
cout << "The chain 'WHEAT->CHEAP->WHEAT->CHEAP' is "
     << (valid ? "" : "NOT ") << "a valid chain." << endl;
```

should result in the output

```
The chain 'WHEAT->CHEAP->WHEAT->CHEAP' is NOT a valid chain.
```

4. Write a Boolean function `find_chain(start_word, target_word, answer_chain, max_steps)` which attempts to find a valid chain beginning with `start_word` and ending with `target_word` in up to `max_steps` steps. If a valid chain can be found, output parameter `answer_chain` should contain the found chain (in the form of a NULL-terminated array of uppercase C-style strings) and the function should return `true`. Otherwise the function should return `false`. For example, the code:

```
const char *answer[100];
bool success = find_chain("HARD", "EASY", answer, 5);
cout << (success ? "Found" : "Could not find")
     << " a chain from 'HARD' to 'EASY' with up to 5 steps"<< endl;

if (success)
  display_chain(answer, cout);
```

should result in the following output (or another valid chain with different links):

```
Found a chain from 'HARD' to 'EASY' with up to 5 steps
HARD
bard
bare
base
ease
EASY
```

For full credit for this question, your function – or helper function if you choose to use one – must be recursive.

(The four parts carry, respectively, 25%, 25%, 20% and 30% of the marks)

Bonus Challenge

For the opportunity to win a mystery prize (which may or may not include extra credit, at the marker's discretion), invent a Doublet puzzle of your own and include it in the Bonus Challenge section of your **main.cpp**. Use your imagination to come up with a good clue. Solutions will be judged using style (e.g. a witty clue or a clever connection between the start and target words) as the primary criterion and solution chain length (longer is better) as the secondary criterion.

What to hand in

Place your function implementations in the file **doublets.cpp** and corresponding function declarations in the file **doublets.h**. Use the file **main.cpp** to test your functions. Create a **makefile** which will compile your submission into an executable file entitled **doublets**.

Hints

1. You will save time if you begin by studying the main program in **main.cpp**, the pre-supplied implementation file **doublets.cpp**, the pre-supplied header files **doublets.h** and **dictionary.h** and the given data file **words.txt**.
2. Do not spend too much time decoding the operation of the Dictionary class in **dictionary.h**. Your time will likely be better spent in understanding how to use the `dictionary_search(...)` helper function declared in **doublets.h** and defined in **doublets.cpp**.
3. Feel free to define any of your own helper functions which would help to make your code more elegant. This applies to **all questions**.
4. **Question 1** will be **much** easier if you exploit the pre-supplied helper function.
5. You should feel free to exploit the answer to any earlier questions when answering a question. This particularly applies to **Questions 3 and 4**.
6. The standard header `<cctype>` contains some library functions that you may find useful when answering **Question 2**. For example, `char tolower(char ch)` returns the lowercase character corresponding to `ch` while `char toupper(char ch)` returns the uppercase character corresponding to `ch`.
7. Using the **-O3** command line option for **g++** turns on many compiler optimisations that will likely speed up the execution of your code. This could be handy when attempting some of the more ambitious Doublets that involve a higher number of steps.
8. Your solution to **Question 4** is required to be **recursive**, i.e. the function (or a helper function if you chose to use one) should call itself. You are permitted to add default arguments if you need to.
9. Your solution to **Question 4** will need to dynamically allocate space for each entry in the array of found strings using the `new` operator. You are not required to free the memory allocated for this purpose.
10. Try to attempt all questions. If you cannot get one of the questions to work, try the next one.