MSc (Computing Science) 2012-2013
C/C++ Laboratory Examination

Imperial College London

Monday 7 January 2013, 15h45 – 17h55

- ☞ You must complete and submit a working program by 17h55.
- ☞ 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 **tube.h**, pre-supplied implementation file **tube.cpp** and to create a **makefile** according to the specifications overleaf.
- ☞ You will find the source files **tube.cpp**, **tube.h** and **main.cpp**, and the data files **map.txt**, **stations.txt** and **lines.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 7 pages.**

Image Credit: `http://www.thepoke.co.uk/2012/03/05/guerillas-on-the-london-underground/`

Problem Description

The London Underground, also known as "The Tube", is one of the world's busiest metro systems, with 1.1 billion passenger journeys p.a.

Most passengers plan their journeys by consulting the Tube map[1], which shows the lines and stations of the London Underground metro network. It is not geographically accurate but is world famous for its elegant angular layout which is similar to that of an electrical circuit diagram. Figure 1 shows a subset of the innermost (Zone 1) stations and lines appearing on a modern Tube map.

Less well-known is Knottenbelt's ASCII Tube map of January 2013 (provided for you in **map.txt**). As shown in Figure 2, it is rather less elegant and even more inaccurate[2]. Critics have been quick to point out that the map needs a key of stations and transit lines to be useful (provided for you in **stations.txt** and **lines.txt** respectively). However, the map does have some unique advantages: it can be stored in under 2Kb (or under 512 bytes when compressed), and can be handily rendered on even the most primitive computing displays and/or printers (e.g. golf-ball printers).

As you know, programmatic support is a great way to add value to digital content. To this end, you are challenged to write some C++ functions which should hopefully help to encourage the adoption of the new ASCII Tube map and assist commuters to plan their journeys.

Pre-supplied functions and files

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

1. `char **load_map(const char *filename, int &height, int &width)` is a function which reads in an ASCII map from the file with name `filename`, sets the output parameters `height` and `width` according to map dimensions, and returns a 2D (`height` × `width`) array of characters.

[1] Initially designed by Harry Beck in 1931 according to Wikipedia.

[2] Astute observers have noted phenomena such as gross distortions of scale, missing lines and stations and the apparent merging of Euston station with Euston Square station.

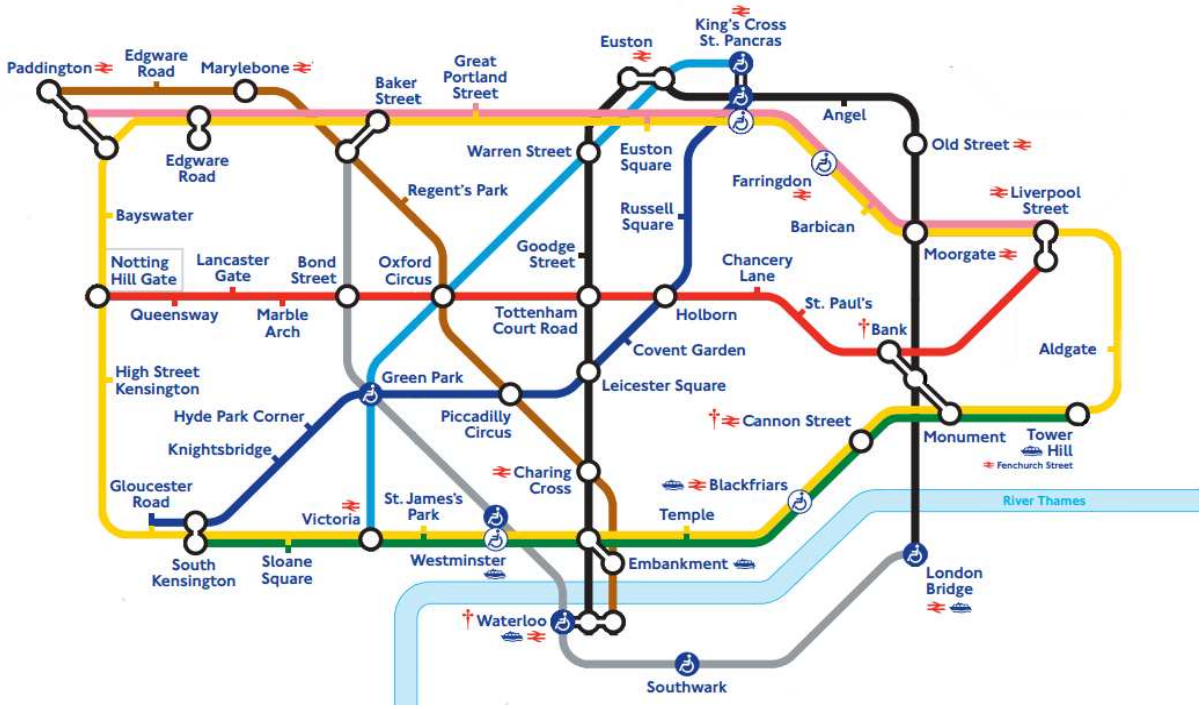

Figure 1: Graphical Tube map of a subset of Zone 1 stations and transit lines (liberally adapted from a contemporary London Underground version).

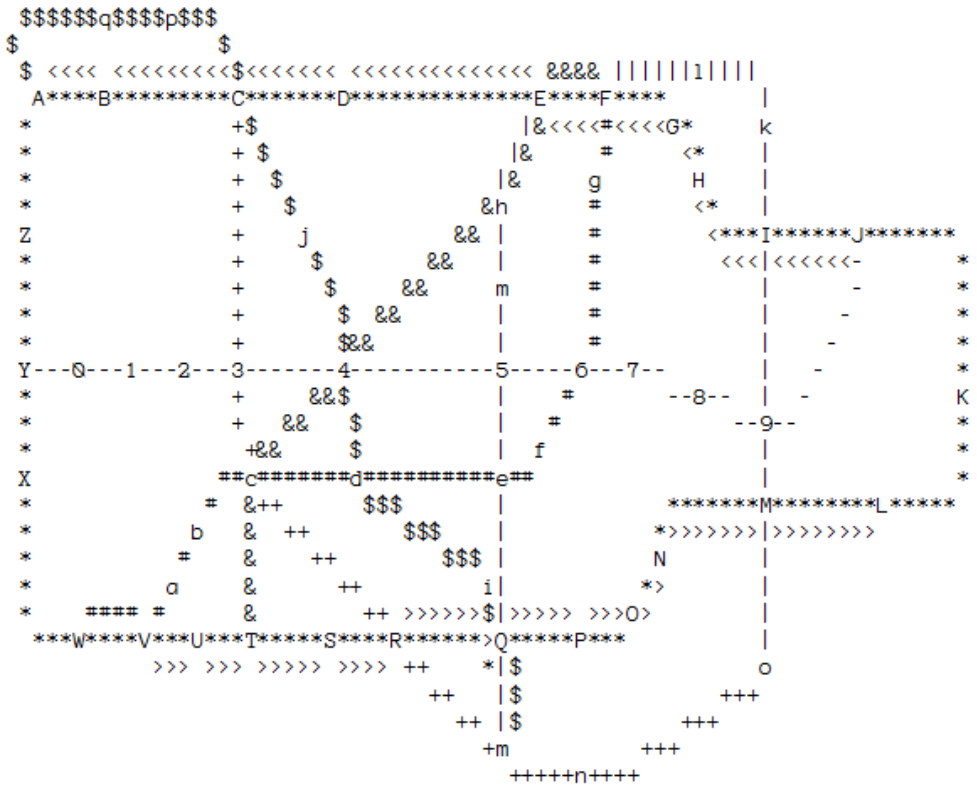

Figure 2: Knottenbelt's ASCII Tube map. Letters and digits indicate stations. Other symbols indicate transit lines.

2. `void print_map(char **m, int height, int width)` displays the ASCII map stored in the 2D (`height` × `width`) array of characters `m`. Row and column numbers are also shown; these can assist you in verifying the map coordinates of stations.

3. `const char *error_description(int code)` provides human readable strings describing the (negative integer) error code passed as a parameter. The error codes themselves are declared in **tube.h**, and will be referred to in Question 3.

4. `Direction string_to_direction(const char *token)` converts a string describing a direction (i.e. "N", "S", "W", "E", "NE", "NW", "SE", "SW") into a corresponding instance of enumerated type `Direction`. The latter is declared in **tube.h** as:

   ```
   enum Direction {N, S, W, E, NE, NW, SE, SW, INVALID_DIRECTION};
   ```

You are also supplied with a main program in **main.cpp** and the three data files mentioned: **map.txt** (containing the ASCII Tube map, as shown in Figure 2), **stations.txt** (containing a mapping from the letters and numbers used to denote stations onto full station names) and **lines.txt** (containing a mapping from line symbols to line names). To be clear, the contents of **stations.txt** are:

```
A Paddington
B Edgware Road (Circle Line)
C Baker Street
D Great Portland Street
... (etc. etc.) ...
q Edgware Road (Bakerloo Line)
```

while the contents of **lines.txt** are:

```
* Circle Line
- Central Line
# Piccadilly Line
& Victoria Line
$ Bakerloo Line
+ Jubilee Line
| Northern Line
> District Line
< Hammersmith & City Line
```

Specific Tasks

1. Write function `get_symbol_position(map,height,width,target,r,c)` that, given a symbol character `target` finds the coordinates (`r`,`c`) of the first occurrence of that symbol on an ASCII map with `height` rows and `width` columns when scanned in a row-by-row fashion. If the symbol is found, the function should return true and set the output parameters `r` (the row number) and `c` (the column number), both of which are indexed starting from 0. If the symbol cannot be found, the function should return false and `r` and `c` should both be set to -1. For example, the code:

   ```
   int r, c;
   bool success =
     get_symbol_position(map, height, width, 'T', r, c);
   ```

   should result in `success` having the value true, `r` having the value 23 and `c` having the value 21, since station 'T' is to be found at map coordinates (23,21). Similarly, the code:

   ```
   int r, c;
   bool success =
     get_symbol_position(map, height, width, 'z', r, c);
   ```

   should result in `success` having the value false, `r` having the value -1 and `c` having the value -1 (since 'z' is not on the map).

2. Write function `get_symbol_for_station_or_line(name)` which, given the input parameter `name` describing the name of a station or line, returns the corresponding map symbol character. If there is no such station or line, return the space character '␣'. For example, the code:

   ```
   cout << "The symbol for Victoria station is '"
     << get_symbol_for_station_or_line("Victoria") << "'";
   cout << endl << "The symbol for the District Line is '"
     << get_symbol_for_station_or_line("District Line")
     << "'" << endl;
   ```

   should result in the output:

   ```
   The symbol for Victoria station is 'T'
   The symbol for the District Line is '>'
   ```

3. Write function `validate_route(map, height, width, start_station, route, destination)` that, given the name of an origin station `start_station` and string `route` describing a passenger journey in terms of the direction taken on the ASCII Tube map at each journey step, determines if the route is valid according to the following rules:

   - The input parameter `start_station` must be a valid station name (as given in **stations.txt**). If not, return error code `ERROR_START_STATION_INVALID`.
   - The input parameter `route` should be a character string describing a sequrence of directions (i.e. "N", "S", "W", "E", "NE", "NW", "SE", "SW") separated by commas. So, for example, a valid route string is `"N,E,SW,N,W"`. If a supplied direction is invalid, return `ERROR_INVALID_DIRECTION`.
   - The passenger journey begins on the ASCII Tube Map at the coordinates of `start_station`, and follows at each journey step the directions given in the route, moving one map square at a time. If this route strays outside the bounds of the map, return `ERROR_OUT_OF_BOUNDS`. If the route strays off a station or line/track, return `ERROR_OFF_TRACK`.
   - Line changes can only take place at stations (since train doors are firmly closed when travelling between stations for safety reasons). If an attempt is made to change lines outside of a station, return `ERROR_LINE_HOPPING_BETWEEN_STATIONS`.
   - An attempt to retrace a journey step outside of a station is not permitted (since trains travel from station to station without reversing). If an attempt is made to do this, return `ERROR_BACKTRACKING_BETWEEN_STATIONS`.
   - The endpoint of the passenger journey should be a station. If not, return `ERROR_ROUTE_ENDPOINT_IS_NOT_STATION`.

   If the route is valid the function should return the number of line changes required to complete the journey as the return value of the function, and assign the output parameter `destination` to be the name of the station at the end of the route. If the route is invalid, the function should return an appropriate error code.

For example, the code:

```
char route[512], destination[512];
strcpy(route, "S,SE,S,S,E,E,E,E,E,E,E,E,E,E,E");
int result = validate_route(map, height, width,
  "Oxford Circus", route, destination);
```

should result in `result` set to `1` (since this is a valid route and 1 line change is required) and `destination` set to "Leicester Square". Several other examples with an indication of expected output are given in **main.cpp**.

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

(The three parts carry, respectively, 25%, 25% and 50% of the marks)

Hints

1. You will save a lot of time if you begin by studying the main program in **main.cpp**, the header file **time.h**, the pre-supplied functions in **time.cpp** and the data files **map.txt**, **stations.txt** and **lines.txt**.
2. The standard header `<cctype>` contains some library functions that you may find useful. For example, `int isalnum(char ch)` returns nonzero if `ch` is a letter or a number.
3. Question 3 will be much easier if you exploit the pre-supplied helper functions.
4. Feel free to define any of your own helper functions which would help to make your code more elegant, particularly when answering Questions 2 and 3.
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).