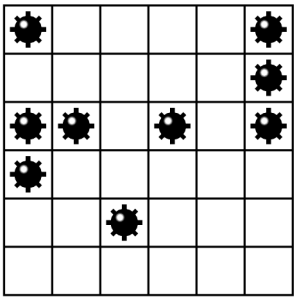
Eric Roberts
CS106B

Handout #17
January 23, 2013

Section Handout #2—ADTs

Problem 1. Using grids (Chapter 5, exercise 10, page 252)

In the game of Minesweeper, a player searches for hidden mines on a rectangular grid that might—for a very small board—look like this:



One way to represent that grid in C++ is to use a grid of Boolean values marking mine locations, where **`true`** indicates the location of a mine. In Boolean form, this sample grid therefore looks like this:

T	F	F	F	F	T
F	F	F	F	F	T
T	T	F	T	F	T
T	F	F	F	F	F
F	F	T	F	F	F
F	F	F	F	F	F

Given such a grid of mine locations, write a function

```
void fixCounts(Grid<bool> & mines, Grid<int> & counts);
```

that creates a grid of integers storing the number of mines in each neighborhood. The neighborhood of a location includes the location itself and the eight adjacent locations, but only if they are inside the boundaries of the grid. The reference parameter **`counts`** is used to store the result. Your job in this exercise is to make sure that it has the same size as the **`mines`** grid and then to assign to each element an integer between 0 and 9. For example, if **`mineLocations`** contains the Boolean grid shown earlier, the code

```
Grid<int> mineCounts;
fixCounts(mineLocations, mineCounts);
```

should initialize **`mineCounts`** as follows:

1	1	0	0	2	2
3	3	2	1	4	3
3	3	2	1	3	2
3	4	3	2	2	1
1	2	1	1	0	0
0	1	1	1	0	0

Problem 2. Using queues (Chapter 5, exercise 13, page 255)

Bob Dylan's 1963 song "The Times They Are A-Changin'" contains the following lines, which are themselves paraphrased from Matthew 19:30:

And the first one now
Will later be last
For the times they are a-changin'

In keeping with this revolutionary sentiment, write a function

```
void reverseQueue(Queue<string> & queue);
```

that reverses the elements in the queue. Remember that you have no access to the internal representation of the queue and will need to come up with an algorithm, presumably involving other data structures, to accomplish the task.

Problem 3. Using maps (Chapter 5, exercise 19, page 257)

In May of 1844, Samuel F. B. Morse sent the message "What hath God wrought!" by telegraph from Washington to Baltimore, heralding the beginning of the age of electronic communication. To make it possible to communicate information using only the presence or absence of a single tone, Morse designed a coding system in which letters and other symbols are represented as coded sequences of short and long tones, traditionally called *dots* and *dashes*. In Morse code, the 26 letters of the alphabet are represented by the following codes:

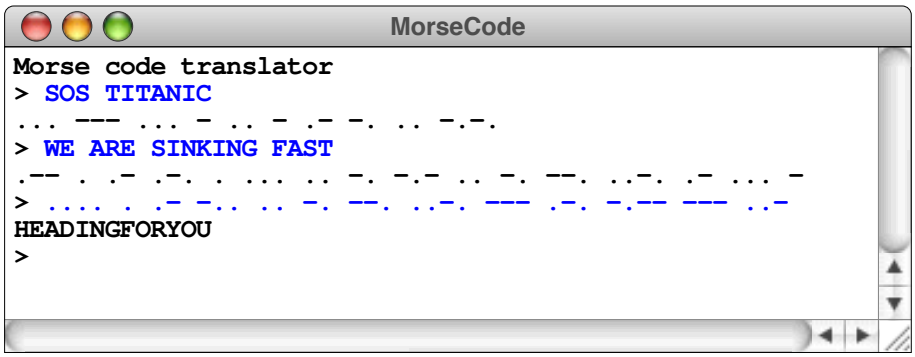
A	• —	J	• — — —	S	• • •
B	— • • •	K	— • —	T	—
C	— • — •	L	• — • •	U	• • —
D	— • •	M	— —	V	• • • —
E	•	N	— •	W	• — —
F	• • — •	O	— — —	X	— • • —
G	— — •	P	• — — •	Y	— • — —
H	• • • •	Q	— — • —	Z	— — • •
I	• •	R	• — •		

Write a program that reads in lines from the user and translates each line either to or from Morse code depending on the first character of the line:

- If the line starts with a letter, you want to translate it to Morse code. Any characters other than the 26 letters should simply be ignored.

- If the line starts with a period (dot) or a hyphen (dash), it should be read as a series of Morse code characters that you need to translate back to letters. Each sequence of dots and dashes is separated by spaces, but any other characters should be ignored.

The program should end when the user enters a blank line. A sample run of this program (taken from the messages between the Titanic and the Carpathia in 1912) might look like this (note that there are no spaces in the Morse-to-letters translation):



Although it is easy to use a `switch` statement to convert from letters to Morse code, converting in the opposite direction requires a map. Given that you need to initialize a map for at least one of the directions, you should try to find a strategy that allows you to use the data in that map for both the letters-to-Morse and the Morse-to-letters translation.

Problem 4. Using lexicons (Chapter 5, exercise 22, page 260)

Section 3.6 defines the function `isPalindrome` that checks whether a word reads identically forward and backward. Use that function together with the English lexicon to print out a list of all words that are palindromes.