Module IN3013/INM173 – Object-Oriented Programming in C++

Solutions to Exercise Sheet 4

1. This is similar to the stack, but since we need to add to one end of the collection and remove elements from the other, we must use a `list` or `deque` instead of a `vector`:

```
template <typename Element>
class queue {
        list<Element> elements;

public:
        bool empty() const { return elements.size() == 0; }

        // Add a new element to the back of the queue
        void enqueue(Element x) {
                elements.push_back(x);
        }

        // Remove an element from the front of the queue
        // require: ! empty()
        Element dequeue() {
                Element x = elements.front();
                elements.pop_front();
                return x;
        }
};
```

   This version has an operation `dequeue` that removes the front element. An alternative design (which may be better) would be to have a function returning the front of the queue, and a procedure to remove it.

2. We need to associate with each word a count, and this is exactly what a `map` is for. Our first cut is

```
map<string, int> occurrences;
string s;
while (cin >> s)
        occurrences[s]++;
```

This sets up the map correctly, but we have no way to print it out. So we add a `vector` of `string`s containing all the unique words we have seen. We want to add each word to this vector, but only if we haven't seen it before. To check this, we can check the `occurrences` map. If the entry for the word if 1 after incrementing, then this is the first time we've seen the word, and we should add it:

```
vector<string> words;
map<string, int> occurrences;
string s;
while (cin >> s) {
        occurrences[s]++;
        if (occurrences[s] == 1)
                words.push_back(s);
}
```

Alternatively, we could test the entry before incrementing:

```
while (cin >> s) {
        if (occurrences[s] == 0)
                words.push_back(s);
        occurrences[s]++;
}
```

Note that if you refer to a mep entry that doesn't exist, it is created with default initialization (e.g. 0 for `int`). We were already using this by incrementing the entry whether or not it already existed.

Yet another alternative is to test for existence of the entry using the `count()` method:

```
while (cin >> s) {
        if (occurrences.count(s) == 0)
                words.push_back(s);
        occurrences[s]++;
}
```

Now we can print the words and their number of occurrences by looping over the vector, giving the complete program:

```
#include <string>
#include <map>
#include <vector>
#include <iostream>

using namespace std;
```

```
int main() {
        vector<string> words;
        map<string, int> occurrences;
        string s;
        while (cin >> s) {
                if (occurrences[s] == 0)
                        words.push_back(s);
                occurrences[s]++;
        }
        for (int i = 0; i < words.size(); i++)
                cout << words[i] << " : " <<
                        occurrences[words[i]] << '\n';
        return 0;
}
```

Note that we index the `words` vector to obtain a word, and then use that as the key of the `occurrences` map.

When we use iterators next session, it will be possible to do this exercise without the auxiliary map.

3. For each word, we need a total and an average, so we introduce a class to hold these values:

```
class Stats {
        double total;
        int count;
public:
        Stats() : total(0), count(0) {}

        virtual void add_value(double v) {
                count++;
                total += v;
        }

        double get_total() { return total; }
        int get_count() { return count; }
        double get_mean() { return total/count; }
};
```

Since we have only a default constructor that initializes each field to te default values of the respective types, we could have omitted it: the compiler would then have generated an equivalent one.

Now we will use a `map` from `string` to `Stats`:

```
map<string, Stats> stats;
```

Now we read words and associated numbers until the end of the input, adding each to the map:

```
string s;
double n;
while (cin >> s >> n)
        stats[s].add_value(n);
```

As in the previous question, we need to keep track of which words we've seen, and we do it in the same way:

```
map<string, Stats> stats;
vector<string> words;
string s;
double n;
while (cin >> s >> n) {
        if (stats.count(s) == 0)
                words.push_back(s);
        stats[s].add_value(n);
}
```

Now we can print the results by iterating over the `words` vector as before:

```
for (int i = 0; i < words.size(); i++) {
        Stats & stat = stats[words[i]];
        cout << words[i] << ": "
                << "total = " << stat.get_total() << ", "
                << "mean = " << stat.get_mean() << << '\n';
}
```

Note the use of the reference `stat` as an alias for `stats[words[i]]` to shorten the output statement. (It is also a bit more efficient, avoiding repeated indexing.)