IN2029 Programming in C++
Solutions to Exercises 4

1. The iterator version is:

```
// the average of the values in a vector
// requires: v.size() > 0
double average(const vector<double> &v) {
    if (v.size() == 0)
        throw domain_error("average of an empty vector");
    double sum = 0;
    for (auto it = v.cbegin(); it != v.cend(); ++it)
        sum += *it;
    return sum / v.size();
}
```

   We can also do this using a ranged **for** loop:

```
// the average of the values in a vector
// requires: v.size() > 0
double average(const vector<double> &v) {
    if (v.size() == 0)
        throw domain_error("average of an empty vector");
    double sum = 0;
    for (auto x : v)
        sum += x;
    return sum / v.size();
}
```

   To make it work with lists, change **vector** to **list**. This works, because the operations used are all supported by lists.

2. The iterator version is:

```
void scale(double s, list<double> &vs) {
    for (auto it = vs.begin(); it != vs.end(); ++it)
        *it *= s;
}
```

   We can also do this using a ranged **for** loop:

```
void scale(double s, list<double> &vs) {
    for (auto &x : vs)
        x *= s;
}
```

The reference **&** here is vital: it means that **x** is an alias for an element in the list, so we are changing that element. Without it, we would be changing a copy, which would then be discarded.

3.
```
// the largest of the values in a vector
// requires: v.size() > 0
double maximum(const vector<double> &v) {
    if (v.size() == 0)
        throw domain_error("median of an empty vector");
    auto it = v.cbegin();
    double largest = *it;
    ++it;
    while (it != v.cend()) {
        if (*it > largest)
            largest = *it;
        ++it;
    }
    return largest;
}
```

As before, to make it work with lists, change each **vector** to **list**.

4.
```
void delete_first_zero(list<double> &vs) {
    auto it = v.cbegin();
    while (it != v.cend()) {
        if (*it == 0) {
            it = v.erase(it);
            return;
        }
        ++it;
    }
}
```

5. This is similar to **maximum** in that we need to treat the first element specially. We can also handle empty lists: they can be left alone.

```
void remdups(list<string> &ws) {
    if (ws.size() > 0) {
```

```
        auto it = v.cbegin();
        string last = *it;
        ++it;
        while (it != v.cend()) {
            if (*it == last)
                it = ws.erase(it);
            else {
                last = *it;
                ++it;
            }
        }
    }
}
```

6. Reading the words is the same as the example in the lecture: if we refer to **`count[w]`** where **`w`** is not yet in the map, a new entry is created, initialized to 0. Then we increment it.

   To make the histogram look nice, we want to add extra padding before the words so that they line up. To do this, we need to to know the length of the longest word, which we get with **`longest_key`**, which iterates through the map.

   Then we perform another iteration through the map to print the entries.

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

using namespace std;

using vec_size = string::size_type;

// length of the longest key in the map
vec_size longest_key(const map<string, int> &m) {
    vec_size max = 0;
    for (const auto &p : m)
        if (p.first.size() > max)
            max = p.first.size();
    return max;
}

// right-align the string in n cells
void pad(const string &s, vec_size n) {
    for (vec_size i = s.size(); i < n; ++i)
```

```
        cout << ' ';
    cout << s;
}

// print n asterisks
void stars(int n) {
    for (int i = 0; i < n; ++i)
        cout << '*';
}

int main() {
    // count of occurrences of each word in the input
    map<string, int> count;

    // read input words, updating their counts
    string w;
    while (cin >> w)
        ++count[w];

    auto width = longest_key(count);
    // write each word and its number of occurrences
    for (const auto &p : count) {
        pad(p.first, width);
        cout << ' ';
        stars(p.second);
        cout << '\n';
    }
    return 0;
}
```

7. For this question we need a vector of numbers for each word. If we refer to **`values[s]`** where **`s`** is not yet in the map, a new entry is created with default initialization, which for vectors sets up an empty vector. Then we add the vew value to it.

   Printing the statistics loops through the container in the same way as before.

```
#include "stats.h"
#include <iostream>
#include <map>
#include <vector>

using namespace std;

int main() {
```

```
    // values associated with each key
    map<string, vector<double>> values;

    // read keys and values
    string s;
    double v;
    while (cin >> s >> v)
        values[s].push_back(v);

    // output keys and statistics
    for (const auto &p : values)
        cout << p.first << ": average = " << average(p.
               second) <<
            ", median = " << median(p.second) << '\n';
    return 0;
}
```