IN2029 Programming in C++
Solutions to Exercises 3

1. This function changes the vector in the same way as **median**, so we shall pass the vector by value. We add the following to `stats.cpp`:

```
// average of all but the highest and lowest scores
// requires: v.size() > 2
double score(vector<double> v) {
    const auto n = v.size();
    if (n <= 2)
        throw domain_error("score of a short vector");
    // sort the whole vector
    sort(v.begin(), v.end());

    // average all but the lowest and highest numbers
    double sum = 0;
    using vec_size = vector<double>::size_type;
    for (vec_size i = 1; i < n-1; ++i)
        sum += v[i];
    return sum/(n-2);
}
```

   (If you used the method from last week's challenge question, there is no need to sort the vector, so we would pass it by **const** reference.)

   Then in `stats.h`, we add the declaration:

```
// average of all but the highest and lowest scores
// requires: v.size() > 2
double score(std::vector<double> v);
```

   Finally, we add code to the **main** function in `main.cpp`:

```
    if (n > 2)
        cout << "score = " << score(values) << '\n';
```

2. These functions only make sense if the vector is non-empty, so we make that a precondition (and test it).

   Our strategy for **maximum** is to keep the largest element seen so far, replacing it if we see a bigger one.

```
// the largest of the values in a vector
// requires: v.size() > 0
double maximum(const vector<double> &v) {
    const auto n = v.size();
    if (n == 0)
        throw domain_error("maximum of an empty vector");
    double largest = v[0];
    using vec_size = vector<double>::size_type;
    for (vec_size i = 1; i < n; ++i)
        if (v[i] > largest)
            largest = v[i];
    return largest;
}
```

Another idea would be to sort the vector and return the last element. This would yield the correct answer, but would be more expensive. Sorting is more time-consuming than a linear scan, and this would also require copying the vector (by passing it by value).

The **`minimum`** function is similar:

```
// the smallest of the values in a vector
// requires: v.size() > 0
double minimum(const vector<double> &v) {
    const auto n = v.size();
    if (n == 0)
        throw domain_error("minimum of an empty vector");
    double smallest = v[0];
    using vec_size = vector<double>::size_type;
    for (vec_size i = 1; i < n; ++i)
        if (v[i] < smallest)
            smallest = v[i];
    return smallest;
}
```

3. We just loop through the possible indices of the vector printing elements:

```
// write vector to an output stream
void write_vector(ostream &out, const vector<double> &v) {
    const auto n = v.size();
    out << "vector:";
    using vec_size = vector<double>::size_type;
    for (vec_size i = 0; i < n; ++i)
        out << ' ' << v[i];
    out << '\n';
}
```

4. Several solutions are possible. Here is one, similar to the code used in Introduction to Algorithms last year.

```
// reverse the contents of a vector
void reverse(vector<double> &v) {
    for (int lo = 0, hi = v.size()-1; lo < hi; ++lo, --hi)
        swap(v[lo], v[hi]);
}
```

Note that the `for` loop here uses a new bit of C++ (and C) syntax: a comma expression like `++lo, --hi` executes the expressions in turn. The value of the whole expression (though it's not used here) is the value of the last expression.

A subtle issue is the choice of `int` here. If we had used an unsigned type, and the vector had size 0, `hi` would be set to a very large number, probably leading to a crash. To use an unsigned type, we would need to guard against this possibility:

```
// reverse the contents of a vector
void reverse(vector<double> &v) {
    using vec_size = vector<double>::size_type;
    if (v.size() > 1)
        for (vec_size lo = 0, hi = v.size()-1; lo < hi; ++
                lo, --hi)
            swap(v[lo], v[hi]);
}
```

This makes use of the fact that a vector of length 0 is its own reverse.

5. To make the header file, we copy all the function signatures, ending them with semicolons. Then we wrap the whole thing in an include guard.

```
#ifndef GUARD_vecutils_h
#define GUARD_vecutils_h

#include <iostream>
#include <vector>

// the largest of the values in a vector
// requires: v.size() > 0
double maximum(const std::vector<double> &v);

// the smallest of the values in a vector
// requires: v.size() > 0
```

```
double minimum(const std::vector<double> &v);

// reverse the contents of a vector
void reverse(std::vector<double> &v);

// read a vector from the input stream
std::vector<double> read_vector(std::istream &in);

// write a vector to the output stream
void write_vector(std::ostream &out, const std::vector<
       double> &v);

#endif
```

6. Like the **maximum** and **minimum** functions, this only makes sense if the vector is non-empty. The logic is similar to exercise 4 from week 2, except that we are iterating over an array.

```
// the longest string in a vector
// requires: v.size() > 0
string longest(const vector<string> &v) {
    const auto n = v.size();
    if (n == 0)
        throw domain_error("longest of an empty vector");
    string w = v[0];
    using vec_size = vector<double>::size_type;
    for (vec_size i = 1; i < n; ++i)
        if (v[i].size() > w.size())
            w = v[i];
    return w;
}
```

The vector parameter **v** is passed by **const** reference, so it won't be copied.

We would also like to avoid copying the string we want to return, returning it by reference, but we can't do that with **w**, because that is a local variable. This version returns it as a reference to an element of the vector **v**:

```
// the longest string in a vector
// requires: v.size() > 0
const string &longest(const vector<string> &v) {
    const auto n = v.size();
    if (n == 0)
        throw domain_error("longest of an empty vector");
    vec_size best = 0;
    using vec_size = vector<double>::size_type;
    for (vec_size i = 1; i < n; ++i)
        if (v[i].size() > v[best].size())
            best = i;
    return v[best];
}
```

We have to add **`const`**, so that the caller cannot change the string we give them, to satisfy the compiler that we are not modifying the **`const`** parameter **`v`**.