Universität Heidelberg
Institut für Technische Informatik

Robert Strzodka
Application Specific Computing

C++ Practice

Term 2019 (Winter)

Exercise 1

- **Submit electronically to Moodle until Tuesday, 2019-10-29, 9:00**
- **A submission consists of a ZIP archive with the complete source code and a PDF document which answers questions from the exercise sheet, briefly explains non-obvious design choices and presents the results. Do not include build artifacts in your submission.**
- **Include names on the top of the sheets.**
- **A maximum of two students is allowed to work jointly on the exercises.**

Reading assignment

Chapters 2, 3, and 8

Preparation

The template project for this exercise is available at https://mp-force.ziti.uni-heidelberg.de/asc/projects/lectures/cpp-practice/WS19/exercises.

Follow the guide in our Wiki to install the libraries `cmakeshift`, `fmt`, and `catch2` using Vcpkg, and to clone and build the project from the exercise template.

If possible, run the measurements on at least two different machines. At least one of the machines should be a `mp-media*` compute node of the ASC cluster (cf. the ASC server hardware overview for a list of available compute nodes and their properties). Be sure to use a release build for performance measurements.

1 Idioms

1.1 Argument passing (8 points)

Function arguments can be passed by value, by reference, or by const reference. Which choice is preferable in each of the following cases? Explain your choice and the semantics and implications of all possible options.

```
double square(double value); // 1.1
double square(double& value); // 1.2
double square(double const& value); // 1.3

double mean(std::vector<double> values); // 2.1
double mean(std::vector<double>& values); // 2.2
double mean(std::vector<double> const& values); // 2.3

// continued on next page
```

```
double logMessage(std::string message); // 3.1
double logMessage(std::string& message); // 3.2
double logMessage(std::string const& message); // 3.3
double logMessage(std::string_view message); // 3.4
double logMessage(std::string_view& message); // 3.5
double logMessage(std::string_view const& message); // 3.6
```

2 Benchmarking and measurements

2.1 Clock resolution (6 points)

For many implementations, `std::chrono::high_resolution_clock::now()` returns a duration of type `std::chrono::nanoseconds`. However, this does not imply that the measurement itself has nanosecond precision.

The following routine from the exercise template attempts to determine the *resolution* of the underlying measurement, i.e. the smallest increment of time that the implementation can reliably report:

```
using Clock = std::chrono::high_resolution_clock;
using DNanoseconds = std::chrono::duration<double, std::nano>;

// bad code
DNanoseconds measureClockResolution_bad()
{
    return Clock::now() - Clock::now();
}
```

Explain why this implementation is flawed, and suggest how it can be improved.

2.2 Microbenchmarking (8 points)

We want to measure the cost of computing a particular floating-point square root using the specialized standard library function `std::sqrt()` and the more general `std::pow()`. Because the runtime of either function might be shorter than the minimal duration resolved by `std::chrono::high_resolution_clock`, they must be executed repeatedly to obtain a precise result.

Explain why the following attempted implementation does not work as expected. Can you suggest how it could be fixed?

```
// bad code
DNanoseconds measureDuration_sqrt_bad(int iterations)
{
    double v = 42;
    auto t0 = Clock::now();
    for (int i = 0; i < iterations; ++i) {
        std::sqrt(v);
    }
    auto t1 = Clock::now();
    return DNanoseconds(t1 - t0) / iterations;
}
```

Hint: *Use Compiler Explorer to look at the code generated by the compiler.*

2.3 Clock latency (8 points)

We want to measure the cost of the function call `Clock::now()` itself. Implement this measurement in the routine `measureClockLatency()` in the exercise template.

3 Iteration and recursion

3.1 Fixed-length sequences (15 points)

Given a certain alphabet, e.g.

```
using FixedAlphabet = std::array<char, 7>;
const auto alphabet = FixedAlphabet{ 'c', 'd', 'e', 'f', 'g', 'a', 'b' };
```

implement two functions with the following signatures:

```
using FixedWord = std::array<char, 3>;

std::vector<FixedWord> fixedSequences(
    FixedAlphabet const& alphabet);

std::vector<FixedWord> fixedSequencesWithoutRepetition(
    FixedAlphabet const& alphabet);
```

where the first computes all possible *sequences* of length 3 using the given alphabet, and the second computes all possible *sequences without repetition*, i.e. sequences of length 3 in which no letter appears twice.

The template project uses the integrated benchmarking support of the Catch2 test framework to measure the runtime for the creation of sequences:

```
BENCHMARK(...)
{
    return fixedSequences(alphabet);
};
```

Report how long your implementation of `fixedSequences()` and `fixedSequencesWithoutRepetition()` takes to run according to this benchmark.

3.2 Flexible-length sequences (35 points)

Improve the functions from Task 3.1 by permitting alphabets and sequences of arbitrary length:

```
using Word = std::vector<char>;
using Alphabet = std::vector<char>;

std::vector<Word> sequences(
    Alphabet const& alphabet, int wordLength);

std::vector<Word> sequencesWithoutRepetition(
    Alphabet const& alphabet, int wordLength);
```

These functions cannot be implemented with nested loops because the length is a runtime parameter. An implementation can instead use either iteration or recursion. Explain the benefits and drawbacks of either approach. Implement the functions using the technique you prefer, but give reasons for your choice.

When producing sequences without repetitions, characters that already occur must be prevented from being used again. Can you do the required lookup in sub-linear time, or even avoid it entirely?

The template project again contains a benchmark that measures the performance for an alphabet of length 9 and word lengths from 1 to 9. Run the benchmark and produce a combined logarithmic plot of the corresponding execution times. Make a second plot with the memory throughput achieved.

Hint: *If the benchmark takes too long to run, you can reduce the number of samples with the `--benchmark-samples <n>` command-line argument. Call `sequences-benchmark --help` to get an overview of the command-line configuration parameters supported by Catch2.*

3.3 Runtime and memory complexity (15 points)

Given an alphabet of size A and a word length L, determine by inspection of your code the number of inner loop iterations performed for a given alphabet and a length. (If you make use of any standard library routines such as `std::find()` or `std::count()`, be sure to consider their runtime complexity, which is specified in the documentation at cppreference.com.)

The number of sequences N_s and the number of sequences without repetition N_u are

$$N_\mathrm{s}(A, L) = A^L \; , \tag{1}$$

$$N_\mathrm{u}(A, L) = \frac{A!}{(A-L)!} \; . \tag{2}$$

Using these and the result of your analysis, determine the number of inner loop iterations required to produce one sequence or one sequence without repetition.

Determine by inspection of your code how much memory is required to produce all sequences, and how much memory is required to produce all sequences without repetition. The computation should cover the memory for the data returned by `sequences()` and `sequencesWithoutRepetition()` as well as the additional memory needed during execution.

Hint: *We are interested in complexity in terms of A and L, not in accurate numbers, hence constant factors may be omitted. This question can be answered without writing code or looking up exact sizes of data types.*