C++ Micro-Projects

Suggestions for Small Exercises, Utilities, Performance Studies, and More

Dipl.-Ing. Martin Weitzel
http://tbfe.de

Technische Beratung für EDV
64380 Roßdorf, Germany

Content

General:

Categories:[*]

[*]: This section currently is *Work in Progress* and will get extended along with my other C++ presentations. Feel free to suggest extensions or improvements on the available channels … (if you are no Robot, you will surely find one :-)) – Martin Weitzel, Roßdorf, July 2015

General Basic C++ Knowledge

Besides the knowledge listed in the *Requirements* section of each project, you should be fluent in the following:

General C++ syntax and concepts, as

- variable definition and scope,
- flow control mechanisms including the basics of exceptions,
- the basics of classes as extensions to data structures
- the (very) basics of inheritance and other OOP core features
- reading input from `std::cin` and writing output to `std::cout`.

Furthermore it may pay to adopt a mild style* of automated testing so that you **do not** need to

- run your program over and over again,
- *manually* typing some test input and
- *visually* checking the output" for correctness.

*: Of course you're welcome to go "full TDD" and write all your tests **prior** to feature implementation.

Micro-Project Rating

Besides focussing on a specific area the suggested micro-projects have a difficulty specified for each project …

… on a scale of four with the following meaning:

trivial:	😉 😉 😉 😉	straight forward, no surprises, go and succeed
simple:	😠 😉 😉 😉	obvious how to approach, if you paid attention
advanced:	😟 😠 😐 😉	requires a small clue or some special insight
challenging:	😟 😱 😐 😎	sometimes end with an aha-experience … [*]

[*]: … "*Oh, that's so simple, really?*" But on the way to that, depending on the approach, a challenging task might also tend to drive some crazy. The following will give an example:

```
 /* A simple word-frequency counting example */
                                                    // 1. add the missing header files (trivial)
                                                    // 2. add the missing `operator<<` for
using WordList = std::map<std::string, int>;        //    `WordList::value_type` (simple .. advanced)
                                                    // 3. enable the alternative way to create output
int main() {                                        //    via `std::copy` (might be challenging)
    WordList wlist;
    using inwords = std::istream_iterator<std::string>;
    std::for_each(inwords{std::cin}, inwords{}, [&wlist](const std::string &w) { ++wlist[w]; });
    for (const auto &e : wlist) std::cout << e << '\n';
    // using outwords = std::ostream_iterator<WordList::value_type>;
    // std::copy(std::begin(wlist), std::end(wlist), outwords{std::cout, "\n"});
}
```

Roll Your Own Micro-Project

If you find nothing among the micro projects which is of interest to you, feel free to roll your own.

Especially, if some topic covered is close to what you may apply in your current or next project,

- why not realize your idea in a "Proof on Concept" form, and
- get some "private coaching" while you are at it?

If you decide to roll your own micro-project you will receive any help to the best of the trainer's ability, but don't expect a perfect sample solution at the end.*

*: And of course, please do not expect or even demand that the trainer will allot time beyond a fair share to your problems, thereby unduly reducing the time available for other participants.

Automated Testing (TDD) Hints

There are many options to automate testing and as many frameworks that support the TDD approach.

To name just a few (sorted in alphabetically, not by personal preference):

- Boost.Test
- CppUnit
- Cute
- Google Test
- Qt's QTestLib
- …

If you are comfortable with one of the above (or equivalent) **just use it** and test your micro-project solution until you get "the green bar".

For all others the following pages outline two "poor men's" approaches:[*]

- Base your Testing on `assert`
- Simplistic Homegrown TDD

[*]: Proposing these alternatives has the only reason that using any of the TDD frameworks from would introduce a dependency that can be avoided with by using a few very basic "C++-Bordmittel".

Assertion Based Testing (1)

Assertion bases testing is based on … `assert`-ions.

The following demonstrates the approach with a test of the faculty function:*

```
#include <cassert> // necessary to access the 'assert'-macro
                   // (as a macro, there is no std::-prefix)

unsigned long long faculty(unsigned long long n) {
    return n ? n*faculty(n-1) : 1;
}

int main() {
    assert(faculty(5) == 120); // 1*2*3*4*5 => 120

    assert(faculty(2) == 2);   // 1*2 => 2
    assert(faculty(1) == 1);   // 1 => 1 (border case)
    assert(faculty(0) == 1);   // => 1 (mathematically correct!)
}
```

*: In an attempt to stay with the most simple of all thinkable solutions, the scheme outlined above applies the "*no news is good news*" principle, i.e. if you run the program and **nothing at all** happens, everything works as expected.

Assertion Based Testing (2)

Making a program "trivially interesting" often involves some input and output. The following applies `assert`-based testing via String Streams.

First the candidate under test:

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

void say_hello(std::istream &in, std::ostream &out) {
    std::string name;
    while (in >> name)
        out << "hello, " << name << '\n';
}
```

Note that the above is not written as `main` program but just as a function.

If course, it could therefore be called that way and tried with manual input and visually checking the response:

```
int main() {
    say_hello(std::cin, std::cout);
}
```

Assertion Based Testing (3)

Of course, what was originally intended was an automated test, possibly supplying different input and comparing the output to an expectation:

```
#include <sstream>

int main() {
    {// ---------------------- test with just one name
        std::istringstream iss("world");
        std::ostringstream oss;
        say_hello(iss, oss);
        assert(oss.str() == "hello, world\n");
    }
    {// ---------------------- test border case (empty input)
        std::istringstream iss("world");
        std::ostringstream oss;
        say_hello(iss, oss);
        assert(oss.str() == ""); // <-- expect not even a newline
    }
    std::cout << "*** ALL TESTS PASSED\n"; // not any longer:
                                           // no news is good news
}
```

*: Note the use of inner blocks to ease adding another test with a simple Copy&Paste.

Simplistic Home-Grown TDD

This approach is a slight extension of the following macro, which turns out to be extremely versatile in small demo programs:

```
#include <iostream>
#define P(what)\
    std::cout << "[" << __LINE__ << "] " #what " --> "\
              << (what) << std::endl
```

The intended use is to print expressions in a "self-documenting" way:

```
#include <algorithm>

int main() {
    auto s = {3, 5, -4, 1, 12};
    auto r = std::minmax(s);
    P(s.first);
    P(s.last);
}
```

The output produced when both fragments are compiled together and run will look as follows:

```
[10] r.first ==> -4
[11] r.second ==> 12
```

(Assuming `P` was used in lines 10 and 11 of the source shown left.)

*: Of course the macro definition may go into a header is included in every source using `P`.

Simplistic Home-Grown TDD (2)

The obvious advantage over Assertion Based Testing is:

- The values are included in the output.

The (also obvious) disadvantage is:

- There is no indication whether the output is what was expected.

The latter can be cured by adding another macro taking the expected value as second argument and showing a different mark (`!=!` instead of `-->`) when the actual value differs from the expected …

```
#define XP(what, expected)\
    std::cout << "[" << __LINE__ << "] " #what\
              << ((what) == expected ? " ==> " : " !=! ")\
              << (what) << std::endl
```

… with the drawback that now only the expected value is shown …[*]

[*]: … though to print the actual value you only need to change back to the `P`-macro … and purge the second argument for the moment … or introduce a pro-forma second argument to `X` too which gets ignored … and of course now any self-proclaimed C language expert world-wide will happily point out that a macro argument should *_never ever_ be used twice … so its time to turn the page once more …

(Not any more so) Simplistic Home-Grown TDD (3)

… and therefore this is what finally connects all the untangled ends:*

```
#define XP(what, expected)\
    do {\
        const auto &w = what;\
        const auto &e = expected;\
        std::cout << "[" << __LINE__ << "] " #what " --> " << w;\
        if (w != e)\
            std::cout << " !=! " <<  e << " <-- (expected)";\
        std::cout << std::endl;\
    } while (0)
```

*: It should be clear that we are quickly approaching the point at which it will probably pay to give up the home-grown approach and switch to a full-blown TDD framework.

(Not any more so) Simplistic Home-Grown TDD (4)

For the sake of completeness here is another test program …

```
#include <algorithm>
#include <sstream>
#include <iterator>

int main() {
    const int x[] = { 1, 5, 5, 8, 1, 12, 1, 5 };
    std::ostringstream oss;
    // do not copy any duplicates from the above sequence
    std::unique_copy(std::begin(x), std::end(x),
                     std::ostream_iterator<int>(oss, " "));
    XP(oss.str(), "1 5 8 12 ");
}
```

… and the message showing how `std::unique_copy` failed:[*]

```
[21] oss.str() --> 1 5 8 1 12 1 5  !=! 1 5 8 12  <-- (expected)
```

[*]: Actually it is not so much `std::unique_copy` that failed – it exactly behaves as specified – but a wrong expectation on the tester's side: not *all* duplicates are skipped over, only *adjacent* duplicates.

Performance Evaluation Hints

A basic way to get timings for performance evaluations using "C++ Bordmittel" only is the `std::clock` library function.

- Its return value is the reflects the CPU time used and
- needs to be divided by `CLOCKS_PER_SEC` to get the time in seconds.

Therefore a very basic helper to time a callable in C++ is the following:

```cpp
template<typename Callable>
double cpu_seconds(Callable fragment) {
    assert(std::clock() != (std::clock_t)(-1));
    const auto start_time = std::clock();
//  -------------------------
    fragment(); // to time
//  -------------------------
    const auto end_time = std::clock();
    return static_cast<double>(end_time - start_time)
         / static_cast<double>(CLOCKS_PER_SEC);
}
```

Performance Evaluation Challenges and Pitfalls

A problem with the technique shown is the resolution of time measuring:

Timing short code sequences might result in "zero CPU time consumed" or give rather jerky results.

To avoid the above, it usually makes sense to

- repeat the commands in a loop, timing the whole loop, and
- divide the total loop time by the runs through the loop.

This leads to two new (major) pitfalls:

- For really short code fragments time measuring may rather time the loop code, not the scrutinized fragment.
- An aggressively optimizing compiler might notice that the loop is "useless" and optimize it away.

[!] Many more pitfalls exist in practice, so be sure to understand the problem at hand, and check all your results for plausibility.

Performance Evaluation – Automatic Loop Sizing

If the problems mentioned so far are solved there is the inconvenience to adapt the repetition count for differently powerful CPUs.

The latter can be solved automatically:

```
template<typename Callable>
std::tuple<std::clock_t, int>
cpu_seconds(Callable fragment, double min = 1.0, int runs = 100) {
    double delta_time;
    int multiply{0};
    do {
        runs *= (multiply ? (multiply = 1) : (multiply *= 2));
        const auto start_time{std::clock()};
        for (auto i = 0; i < runs; ++i)
            fragment(); // to time
        const auto end_time{std::clock()};
        delta_time = static_cast<double>(end_time - start_time)
                   / static_cast<double>(CLOCKS_PER_SEC);
    } while (delta_time < min);
    return {delta_time, runs};
}
```

*: To further improve measuring and accounting process time, Boost.Chrono may be of interest to you because of some extensions it provides over the Chrono Library as standardised with C++11.

STL Specific Micro-Projects

Following are some suggestions for micro-projects involving the STL.

Expected for all projects there is some

- Basic C++ Knowledge, and
- **in addition** the instantiation syntax of templates.[*]

Given that, you may select from the following categories:

- Container Centric
- Algorithm Centric
- Extension Centric
- Miscellaneous Others

---

[*]: **Not:** How to define templated containers and algorithm though you will probably gain some insight by working on the projects.

Container Centric STL Micro-Projects

1. Implement a *Word Frequency Analysis* with an `std::map`
2. Implement a *Word Frequency Analysis* with an `std::multiset`
3. Transform a Text (File) into a *Word Position Table*
4. Transform *Word Position Table* back into a Text (File)

***Word Frequency Analysis* with `std::map`**

Difficulty

trivial: 😉 😉 😉 😉

Example

From the following input …

```
beware the Jabberwock my son
beware the Jubjub bird and shun
the frumious Bandersnatch
```

… produce that output:

```
Bandersnatch: 1
Jabberwock: 1
Jubjub: 1
and: 1
beware: 2
bird: 1
frumious: 1
my: 1
shun: 1
son: 1
the: 3
```

Requirements

Understand the concept of an STL map, used as:

```
std::map<std::string, int> word_freq;
```

Especially understand what the following expression returns, in both cases, for an existing and a non-existing key used in the square brackets:

```
… word_freq[w] … // assume w is of type std::string or compatible
```

Steps to Solution

1. Implement the input loop:
    - extract words with `operator>>` from a stream into a string until EOF
    - update the count per word
2. Implement the output loop:
    - traverse all entries of `word_freq`
    - print the word and its frequency (one per line)

Questions

In which order do you expect the words to be printed?

How does this depend on the specify type of map used?

Advanced Aspects

How could the sort order of the words be changed?

How could the words be reproduced in "insertion order" (of its first occurrence)?

Word Frequency Analysis with `std::multiset`

Difficulty

simple: 😫 😉 😉 😉

Example

From the following input …

```
beware the Jabberwock my son
beware the Jubjub bird and shun
the frumious Bandersnatch
```

… produce that output:

```
Bandersnatch: 1
Jabberwock: 1
Jubjub: 1
and: 1
beware: 2
bird: 1
frumious: 1
my: 1
shun: 1
son: 1
the: 3
```

Requirements

Understand the concept of an STL multiset, used as:

```
std::multiset<std::string, int> word_freq;
```

Understand (or look-up)

- how the number of indices with the same value can be determined, and
- how to skip over all but the first in a loop.

Steps To Solution

1. Implement the input loop:
    - extract words with `operator>>` from a stream into a string until EOF
    - update the count per word

2. Implement the output loop:
    - traverse the entries of `word_freq`
    - determine the frequency of a word (by the number of entries existing for it)
    - print the word and the number of occurrences (in a single line)
    - skip over all the following entries for that word

Questions

In which order do you expect the words to be printed?

How does this depend on the specify type of map used?

Advanced Aspects

How could the sort order of the words be changed?

If no particular sort order is required, were it possible to use an `std::unordered_multiset`?

(Why? Isn't it that the order of traversal is arbitrary – so, if you tried, did it possibly work only by chance?)

Text to *Word Position Table*

Difficulty

simple to *advanced:* 😠 😉 😉 😉 / 😟 😠 😐 😉

Example

From the following input …

```
beware the Jabberwock my son
beware the Jubjub bird and shun
the frumious Bandersnatch
```

… produce that output:

```
Bandersnatch: 4.2
Jabberwock: 2.2
Jubjub: 3.2
and: 3.4
beware: 2.0 3.0
bird: 3.3
frumious: 4.1
my: 2.3
shun: 3.5
son: 2.4
the: 2.1 3.1 4.0
```

Requirements

Understand that

- input analysis **requires two nested loops**

and how to

- extract a full from a stream into an `std::string` using`std::getline`,
- extract a world from stream into an `std::string` using `operator>>`,
- turn an `std::string` into an `std::istringstream`.

Also understand what in the following fragment is acceptable inside the square brackets and what is the (type of) the resulting expression.

```
struct line_pos {
    int line, pos;
    line_pos(int line_, int pos_) : line(line_), pos(pos_) {}
};
std::map<std::string, std::vector<line_pos>> word_tab;
…
//               vvv------- what type can go here?
    … word_freq[ … ] …     // and
//    ^^^^^^^^^^^^^^------ what is the result of this?
```

Steps To Solution

1. Implement the two nested input loop:

   - the outer loop needs to read full lines until EOF
     - while taking care of a line count (= total number of lines read so far)
   - turn these line into an `std::stringstream` from which
   - the inner loops extracts all contained words
     - while taking care of the position count (= number of words read from **this** line)

2. Implement the two nested output loop:

   - traverse the entries of `word_tab`
     - print the word (element `first` of `std::pair<std::string, … >`)
   - in a **nested** loop traverse the line and position numbers (element `second` from `std::pair< … , std::vector<line_pos>>`)
     - print all line and position numbers following the word they belong to …
     - … then add a line end (`\n` or `std::endl`)

Questions

Might it make sense to use `std::copy` as part of the solution?

In the input loop?

In the output loop?

Word Position Table **to Text**

Difficulty

simple to *advanced:* 😣 😉 😉 😉 / 😟 😣 😐 😉

Example

From the following input …

```
Bandersnatch: 4.2
Jabberwock: 2.2
Jubjub: 3.2
and: 3.4
beware: 2.0 3.0
bird: 3.3
frumious: 4.1
my: 2.3
shun: 3.5
son: 2.4
the: 2.1 3.1 4.0
```

… produce that output:

```
beware the Jabberwock my son
beware the Jubjub bird and shun
the frumious Bandersnatch
```

Requirements

Some experience with parsing input (like shown), e.g. by

- splitting an `std::string` at a given point,
- **or** combining `std::istringstreams` with nested `std::getline` calls[*], using separators different separators (`:` and `.`),
- **or** using regular expressions.

Understand that – in the general case – you need to have seen (and stored) **all input** before you can produce output, as the first word might come last.

More depends on the solution, especially the data structure you chose to store the input until output is produced.

The least amount of work will probably be necessary for this:

```
std::map<int, std::map<int, std::string>> word_tab;
```

[*]: And therefore understand that `std::getline` should rather have been named: `get_until_separator_and_swallow_the_latter`

Steps To Solution

1. Implement reading input – **this is the more complicated part**
   - details depend on the way you chose to parse the lines
2. Implement generating output – this is the easy part
   - with the storage structure proposed, this can be solved with two simple (nested) range-`for` loops

Questions

How robust is your solution against "invalid input"?

How could a minor addition and modification to the storage structure avoid storing repeated words many times?

---

*: Some hints (if you need):
1. Think along the lines of "pointers" but chose something "more in the spirit of the STL".
2. Each word needs to be stored exactly once (well, about that is the whole idea of the improvement).
3. Look-up what is the return value when inserting an element into an associative container.
4. … no, NO, **NO!** – leave it, more hints would take out all of the fun :-)

Algorithm Centric STL Micro-Projects

Small Challenges (and Performance Testing)

Difficulty

trivial to *challenging:* 😉 😉 😉 😉 / 😟 😱 😐 😎

Requirements

Online or offline reference for STL algorithms.

(If considered necessary, hints are given with the individual steps.)

Often the following challenges involve some practical performance testing. It might pay to have a cursory look on the page that explains the basics of Getting Timings in an automatised way.

The solution* itself may often be one-liner or close, so the "art" rather is to select the STL algorithm most appropriate to solve the task at hand.

*: This does not include the frame to set up the test data. But that part may often be reused from a former challenge. Also understand that the goal is **not** to come up with a clever but overly cryptic "one-liner". If more lines of code – say a local temporary or type alias – will make the solution better comprehensible, then add it!

Organisation into Groups

As the following are rather "Nano-" or "Pico-Projects" they are organised slightly different as the other "Micro-Projects", and furthermore organised as sub-groups.

- Modifying Sequence Operations
- Non-Modifying Sequence Operations
- Partitioning and Sorting Operations
- Binary Search (on Sorted Ranges)
- Set Operations (on Sorted Ranges)
- Heap Operations
- Minimum/Maximum Operations
- Numeric Operations
- C-Library: `qsort` and `bsearch`

Modifying sequence operations are put first, as you will learn to create large data sets, useful for some performance measurements.

After that goal is achieved, it is suggested **not** to work through all the groups linearly, but "move around" within **and** between the groups.

Modifying Sequence Operations

C++11 added the algorithm `std::copy_if`. But the fact that this was "missing" from C++98 was not that much a problem, as what it does can easily be done by `std::remove_copy_if` too.

How?

Demonstrate with an example program!

Can you spot other "redundant functionality" in that sense, when skimming the content of the Algorithm Library?

(The last question is posed not only with respect to the *modifying sequence operations*, as covered in this section, but all of the algorithm library.)

Modifying Sequence Operations (cont.)

Fill different kinds of containers of `int`-values using `std::iota`[*]

Is the difference in speed for different container types measurable?

- Which is the least size that shows a (substantial) difference?
- Which container type performs best, which worst?
- Given there are 5000 elements, what is the order of magnitude for the difference between the best and the worst(1:2, 1:10, 1:100 …)?

Name an algorithm that allows a more flexible way to generate the initial content of an STL container, so that the container may be filled with:

- multiples of three
- random numbers
- primes

Demonstrate at least one of the above with the chosen algorithm and get the timing.

[*]: As `std::iota` became part of the library with C++11, it may not be available with some older library version. In this case: skip hat step or find some other algorithm set may be used.

Modifying Sequence Operations (cont.)

Given a random number generator with an *iterator style interface*,[*] how could you use `std::copy_n` to fill a container with `N` random values?

As `std::copy_n` wasn't part of C++11: what were the difficulty if the same had to be done with `std::copy`?

Why does an iterator style interface

- not fit to the requirements of `std::generate` / `std::generate_n`, and
- how could you transform that interface "on the fly" with a Lambda?

---

[*]: A very bare-bones implementation of such a `RandomIterator` class might look as follows (`std::rand` for simplicity and despite its well-known deficiencies):

```
// random number generator with iterator-style interface
#include <cstdlib>
#include <iterator>

struct RandomIterator
        : std::iterator<std::input_iterator_tag, int> {
    RandomIterator operator++() {
        return *this;
    }
    int operator*() const {
        return std::rand();
    }
};
```

Modifying Sequence Operations (cont.)

From an `std::vector<unsigned int> x`, filled with ascending values from 0 to 9999, generate a second `std::vector<float> y`, filled with the square roots of these numbers in `x`.[*]

Do the same for a vector filled with random positive numbers.

[!] You may use this for generating large sequences of floats as test data, but be sure to understand that such sequences are not really random in their mathematical distribution.

So if the object under tests may be effected by the mathematical distribution don't draw any conclusions from simple tests with floats generated in that way.

Instead you may want to look at the Random Number Generation (sub-) library introduced with C++11.

[*]: Hint, if you need one: make friends with `std::transform`.

Modifying Sequence Operations (cont.)

Given the following scenario: there are two containers of the same size …

… one holding instances of `MyClass` …

```
std::list<MyClass> objs;
… // fill with instances, say
  // a, b, c ...
```

… the other simply holding `int` values:

```
std::vector<int> args;
… // fill with values, say
  // 12, 17, 108 ...
```

What is the most direct way (avoiding any loops, and also `std::for_each`)

- to call the member function `MyClass::foo(int)`
  - for the given objects with the given arguments,
  - and the return values
    - inserted to a stream `os`,
    - followed by a slash,
- i.e the equivalent of the following:*

```
os << a.foo(12) << '/' << b.foo(17) << '/' <<  c.foo(108) << '/' …
```

*: Hint, if you need one: make friends with `std::transform`.

Modifying Sequence Operations (cont.)

Create a sufficient amount of test data for points on a 2D-surface, specified in Cartesian coordinates, as sketched below:

```
struct CartesianPoint { double x, y };
std::vector<CartesianPoint> cc_points(20);
std::generate_n( … , … , [] { const auto x{(1e3*std::rand())/1e5};
                              const auto y{(1e3*std::rand())/1e5};
                              return … x … y …;
                         });
```

Given the above exists and is properly initialized, how to create a second vector, holding equivalent points in polar coordinates?*

```
struct PolarPoint { double dist, angle; };
std::vector<PolarPoint> pc_points;
```

If you have ideas for different ways to approach the task at hand, try all and compare performance for a sufficiently large amount of test data.

*: If your school days are too far away to remember the computation of polar coordinates from Cartesian, here is a quick refresher: `dist = std::sqrt(x*x + y*y); angle = std::atan2(y, x);`

Modifying Sequence Operations (cont.)

Device a scheme to

- turn the content of a `std::forward_list<unsigned long long>`
- into a text representation as `std::list.*

If you have a translation environment based on C++98 only,

- base your solution on `std::copy` and an `std::sstream`, accessed via `std::istream_iterator`-s and `std::ostream_iterator`-s.

If you have a translation environment based on C++11

- base a (second) solution on `std::transform` plus `std::to_string`,
- and do a comparison which solution performs better.

Are the comparison results plausible?

(Depending on whether the results are plausible to you or not: Explain to yourself or a colleague why they are or why you question the plausability!)

*: The specific container types are not what is of interest here, it is the challenge to transformation of a native type to its text representation.

Modifying Sequence Operations (cont.)

Fill a vector of notable size with random numbers in the following ways and get the timings:

- Simply add new elements at the end.
- **First** set the number of vector elements accordingly (using `resize`).
- **First** set the vector data space accordingly (using `reserve`).

Try to get the timing for the *sizing* operations in the second and the third case separately from the *filling* operation and chose a size at which the differences become clearly noticeable.

Are the differences in the results plausible?

Explain why!

Repeat the above for

- an `std::unordered_multiset` (i.e. no resizing but space reservation), and
- an `std::array` of fixed size (i.e. no resizing and no space reservation).

Again: Are the results plausible and can you explain why?

Modifying Sequence Operations (cont.)

Device a technique with which you can generate a number of random integer values with a given percentage of duplicates,* i.e. if "30%" duplicates were chosen and then

- `std::sort` followed by `std::unique` is applied,
- it's size should shrink by 30%.

Test data with a known percentage of duplicates can be helpful in comparing built-in sets to Set Operations (on Sorted Ranges).

- If you have been redirected to here from a micro project requiring that kind of test data, continue there.

- Otherwise just verify that the test data is correctly created.

*: Some hint, if you need one: create a unique, ascending sequence, then add the percentage chosen by selecting values at random positions **within** that sequence to its end, and finally "shuffle". Also understand that there are more properties of duplicates with a potential impact on the result of performance evaluations, like the distribution of duplicates.

Modifying Sequence Operations (cont.)

Generate a large random sequence, then compare performance for removing half of its values[*] using

- either the container specific member function `remove_if`.
- or the generic algorithm `std::remove_if`.

If you use the generic algorithm on a resizable container, also get rid of the "(now) garbage at the end".

Do this for a selection of STL containers you are interested in.

(Doing it for all eight containers plus native arrays is a task requiring great diligence, so you may well stop when you feel you have gathered enough insight.)

[*]: Hint: if it can be assumed that even and odd numbers are equally distributed, just remove the value with the lowest bit set.

Modifying Sequence Operations (cont.)

From the following ways to copy the content of a container, which do you think is fastest?

```
constexpr std::size_t N = …; // make an appropriate choice
std::vector<double> v(N);    // maybe fill with std::generate_n ??

/*1*/ std::vector<double> v1;
      std::copy(v.begin(), v.end(), std::back_inserter(v1));
/*2*/ std::vector<double> v2; v2.resize(v.size());
      std::copy(v.begin(), v.end(), v2.begin());
/*3*/ std::vector<double> v3; v3.reserve(v.size());
      std::copy(v.begin(), v.end(), std::back_inserter(v3));
/*4*/ std::vector<double> v4(v.begin(), v.end());
/*5*/ std::vector<double> v5(v);
/*6*/ std::vector<double> v6 = v;
/*7*/ std::vector<double> v7; v7 = v;
/*8*/ std::vector<double> v8(std::move(v)); // no later use of v !!
```

Do **not try** (right now) – think about it and give a **guess**.*

*: Of course, after guessing you may practically prove (or disprove) your answer. (Would you have answered differently for an `std::array`? Or if the element type were `std::string`? If `v` were defined `const`?)

Modifying Sequence Operations (cont.)

Determine the speed (difference) of `std::copy` and `std::move` for containers filled with

- `int`-s,
- `double`-s, and
- `std::string`-s.

It will most probably not make any difference whether the containers are filled with random values, or all values are the same.*

Try it for some different containers, at least `std::array`, `std::vector`, and `std::list` …

Furthermore, what difference makes it, if instead of the (generic) `std::copy` or `std::move` algorithms

- `std:move_backward` or `std::move_backward` is used,
- the assignment operation for the container class is used,
- the source container `s` is moved to the target `t`, i.e. `t = std::move(s)`.

*: But of course, as a diligent and *true STL scientist*, you might not rely on your assumptions or "gut feelings", you might test it too …

Modifying Sequence Operations (cont.)

To correctly copy or move **overlapping regions** within a container

which of

- `std::copy` vs.
- `std::copy_backward`,

and

- `std::move` vs.
- `std::move_backward`

needs to be used, depending on

- the location of the source (region)
- with respect to the destination position?

If you came here from
[adding a `universal_move` / `universal_copy` algorithm]
go back now.

Modifying Sequence Operations (cont.)

Given three iterators `from`, `upto`, and `dest`, and assuming

- `from` closer to the container front as `upto`, and
- `upto` closer to the container front as `dest`, …

```
std::rotate(from, upto, dest);
```

… depict graphically the effect of the code fragment shown left.*

Then for the following, slightly changed assumptions

- `dest` closer to the container front as `from`, and
- `from` closer to the container front as `upto` …

```
rotate(dest, from, upto);
```

… again depict the effect of the code fragment shown left.

If you came here from [adding a `slide` algorithm], go back now.

*: Assume the layout of an `std::array` or `std::vector` with elements adjacent to each other, draw the "state before" and the "state after" side by side, colorize the area between `from` and `upto` in the "state before", the colorize the same elements in their new position, in the state after.

Non-Modifying Sequence Operations

Considering `std::for_each` what are the advantages over:

- A "hand-written", index-based loop?
- A "hand-written", iterator-based loop?
- A C++11 range-`for` loop?

Would your answer be different if:

- Depending on the exact container type?
- If const-correctness needs to be ensured?
- If the loop does
  - not visit all elements of a container, or
  - needs to run "back to front"?
- Depending on whether the loop body
  - has to go in a functor (only option for C++98 and C++03), or
  - may alternately use a lambda (since C++11)?
- The loop body needs
  - access some variables in the callers scope (read-only), or
  - has to transfer a result back to the callers scope?
- If performance* is absolutely crucial?

---

*: Concerning performance, maybe do some measurements, comparing `std::for_each` with its alternatives.

Non-Modifying Sequence Operations (cont.)

Consider the group `std::all_of`, `std::any_of`, and `std::none_of`.

Is it that actually all three are required?

Can you demonstrate how one of it might be substituted by some other?

Might even each of the three substitute both of the others?

- If this is **not** so, which one(s) is/are the indispensable?
- If so, why have all three?
  - For performance?
  - For readability?
  - For both reasons?

Can you demonstrate – via taking timings – any "short-cut" effect, i.e. that these algorithm run faster (or slower) depending on the particular data?

Non-Modifying Sequence Operations (cont.)

Do a performance comparison for `std::count_if`, depending on how the predicate is handed over:

- via a lambda;
- via a functor and the relevant `operator()` overload defined
  - implicitly inline, or
  - explicitly `inline`;
- with a function pointer and the function implementation
  - in the same translation unit,
    - without the keyword `inline`, or
    - with the keyword inline, or
  - in a different translation unit (so "silent inlining by the compiler is made impossible).

Compare for constant conditions (e.g. count all elements less than 12) and for the case (say: the limit 12) comes from a variable of the calling scope.

To complete the set of comparisons, also compare with a case for which the plain `std::count` (comparison with fixed value) would have sufficed.

Non-Modifying Sequence Operations (cont.)

Is the following code "correct" considering the leeway the STL specification gives to implement `std::find_if`?

```
// find first value exceeding the sum of all the previous ones:
std::vector<int> v;
… // assume v filled somehow
assert(v.size() >= 1)
int sum = v.font();
auto pos = std::find_if(v.begin()+1, v.end(),
                        [&sum](int e) { return (e > sum)
                                            ? true
                                            : (sum += e, false);
                        });
```

Will your answer change if not the first of these values is looked up, but all are counted with `std::count_if`?

```
auto cnt = std::count_if(v.begin()+1, v.end(),
                         [&sum](int e) { const auto res{e > sum};
                                         sum += e;
                                         return res;
                         });
```

Non-Modifying Sequence Operations (cont.)

(i) The following is advanced in so far as it expects familiarity with C++11 Multi-Threading, at least to the degree that the use of `std::async` is understood.*

Given your program executes on a multi-core CPU, try to beat the STL implementation of `std::count` performance-wise:

- Split the work to be done to two or four distinct data blocks (ideally chose this number according to the CPU capabilities),
- start the functions asynchronously,
- then collect (and finally add) the results.

*: Of course, if you are hot now on trying to beat the STL library performance on a multi-core CPU, that you are willing to acquire the necessary knowledge right now, feel free to do so. Especially as using `std::async` is not **that** hard and you need to know close to nothing about all the intricacies that usually go with multi-threading.

Non-Modifying Sequence Operations (cont.)

Compared to the following code fragment …

```
struct LessThanLast {
    std::string &last;
    LessThanLast(const string &last_) : last(last_) {}
    bool operator(const std::string &e) {
        const std::string t = last; last = e; return (e < t);
    }
};
… // assume properly initialised std::vector<std::string> v
std::string last{};
auto pos = std::find_if(v.begin(), v.end(), LessThanLast(last));
```

… first guess, then try what gives the bigger performance improvement:

- while **keeping** the implementation of `LessThanLast::operator()` …
    - … replace the functor with an equivalent lambda;
    - … replace `std::find_if` with an equivalent hand-written loop;
- or **changing** the implementation of the functor to:

```
        const bool r = (e < last); last = e; return r;
```

Non-Modifying Sequence Operations (cont.)

First: Understand the difference between

- "finding" – i.e. the `std::find…`-algorithm family and
- "searching" – i.e. the `std::search…`-algorithm family.*

Then: Determine by a practical examination whether there is (still) a substantial difference in speed, so that it could make sense to

- first check if the sequence to *search* for has only one member, and
- then eventually prefer *find* over it?

(Maybe given you have determined in advance that 90% of your searching through the haystack is a for a needle of just one element.)

If you have installed Boost.Algorithm you may include in your research also implementations according to Boyer-Moore, Boyer-Moore-Horspool, and Knuth-Morris-Pratt.

*: You see, it is not quite the difference of the colloquial use, but depending on the outcome of the above, you might have an aide-memoir: *Finding* is better than *Searching*.

Non-Modifying Sequence Operations (cont.)

How does comparing two ranges for equal content with a generic algorithm from the `std::equal…`-family perform in comparison to `operator==`, applied to the container itself?

Direct container comparison (`operator==` forwards to comparing element by element) …

```
std::vector<int> x, y;
… // fill with data
if (x == y) …
```

… and the generic STL algorithm:

```
std::vector<int> x, y;
… // fill with data
if (equal(x.begin(), x.end(),
          y.begin(), y.end());
```

Repeat the comparison for an `std::deque` and an `std::list`.*

First think about this:
Which obvious shortcut exists in the implementation of `operator==` for some container class … (to hint further: except for `std::forward_list`)?

Then: Design your test cases to show that effect (too)!

*: But also be sure to understand that the (probably much worse) results for the `std::list` come from bad cache-hit-rates, caused by its non-contiguous memory layout.

Non-Modifying Sequence Operations (cont.)

Given the following simple *Iota Generator with an Iterator Interface* …

```
class IotaGenerator
    : public std::iterator<std::input_iterator_tag, int> {
    int value;
public:
    IotaGenerator(bool zero_origin = true)
        : value(zero_origin ? 0 : 1) {}
    IotaGenerator operator++() {
        return *this;
    }
    int operator*() const {
        return value++;
    }
};
```

… how could you use this to check whether some `std::vector<int>` holds consecutively increasing values 0, 1, 2, … (up to its size less 1)?

- How would you chose 1-origin instead of 0-origin?*

*: Only as a footnote as it is an aside to the algorithms, covered here: which modifications would make sense to give the `IotaGenerator` an even broader applicability, say generate a sequence from any start value with any step width, or even with any expression calculating the next value?

Non-Modifying Sequence Operations (cont.)

The `std::equal…`-family has four members total. Decide which one would you chose to solve the following problems to compare …

- … different containers of the same size, with the same element type;
- … containers of the same type, including the same element type, but of different size, for an equality within the limits of the shorter one;
- … containers of same type and with the same element type, the second possibly being shorter;
- … `std::list<std::string>` containers of equal size, ignoring differences with respect to upper/lower case;
- … `std::vector<int>` with `std::vector<double>`;
- … `std::array<double>` with `std::istringstream` holding `double` values in text form;
- … `std::vector<const char *>` with `std::vector<const std::string>` (based on the content of the strings if interpreted according to the C-rules).
- … `std::array<double>` with an array of the same size, assuming the content of the strings is convertible to double;
- … as before but taking empty strings to represent zero.

*: Of course it would be nice if you defend your decision by providing some SSCCEs, and in case there are different equally simple solutions, if you point out which one is more performant.

Partitioning and Sorting Operations

How would you sort …

- … an `std::array<int>` into decreasing order, using …
  - … a comparison function (i.e. function pointer)?
  - … a comparison functor?
  - … a comparison lambda?
  - … nothing of the above but "C++98 Bordmittel" only?
    (Hint: see `header <functional>`)
- … an `std::vector<const char *>` (assumed to hold "classic C-strings") by their native character set order, i.e. like `std:strcmp` compares?*
- … an `std::vector<std::string>` ignoring differences in case?
- … an `std::forward_list<double>` into ascending order?
- … an `std::set<std::string>` into descending order?
- … an `std::array<int>` as shown in the example below:
  - 5, 7, -4, 33, 112, -5, 17, 12, -1, 33, 1024 (before sorting)
  - 5, 7, 12, 17, 33, 33, 112, 1024, -5, -4, -1 (after sorting)

---

*: Note that this is slightly underspecified, as it is not said whether the order of code points or code units are to be considered. The good news is: for UTF-8 coded `std::string`-s it doesn't matter - both orders are the same. For UTF-16 coded `std::wstrings` this is only true for characters in the BMP (assuming `wchar_t` having 16 bits, what is the usual choice in MS-Windows and Java).

Partitioning and Sorting Operations (cont.)

Create a sufficient amount of test data for points on a 2D-surface and …

```
struct Point { int x, y; };
std::vector<Point> points;
```

… compare the different effects – **including performance** – of:

```
// (1) sort by primary and secondary criteria in one step
std::sort(points.begin(), points.end(),
          [](int lhs, int rhs) { return ((lhs.x < rhs.x)
                                      || (!(rhs.x < lhs.x)
                                         && (lhs.y < rhs.y))); });

// (2) sort by primary criteria first, then sort secondary
std::sort(points.begin(), points.end(),
          [](int lhs, int rhs) { return lhs.x < rhs.x; });
std::sort(points.begin(), points.end(),
          [](int lhs, int rhs) { return (lhs.y < rhs.y); });

// (3) as before, but STABLE sort by the second criteria
std::sort(points.begin(), points.end(),
          [](int lhs, int rhs) { return (lhs.x < rhs.x); });
std::stable_sort(points.begin(), points.end(),
          [](int lhs, int rhs) { return (lhs.y < rhs.y); });
```

Partitioning and Sorting Operations (cont.)

Do a performance evaluation of `std::sort` vs. `std::stable_sort` using

- test containers with a varying percentages of elements comparing equal
  - from all elements unique
  - to all elements equal
- and element types with different "costs" for swapping.

To prepare your analysis, you might **think about swapping costs** first, or even run some isolated tests in advance, so that the degree to which swapping costs vary in your test-setup becomes clear.*

*: As a hint for the lazy (or those who want to hurry to the core task): Choosing `std::vector<int>` **as element type(!)** for the sorted container, vs. choosing `std:array<int>` will dramatically vary the swapping cost, easily controlled by the size of those **element** containers. Also, the comparison cost can be controlled by defining an comparison operation that compares a smaller or larger initial part of the element containers.

Partitioning and Sorting Operations (cont.)

For a large `std::vector<int>` (say hundred-thousand elements or more), what kind of partial sort takes the most time, and what the least:*

Sorting **smallest values to front**, so that

- the first 10% of the container
  - is still **not fully** sorted, but
  - holds **no larger** elements as the remaining 90%;
- the first half of the container
  - is still **not fully** sorted, but
  - holds **no larger** elements as the second half;
- the first 90% of the container
  - is still **not fully** sorted, but
  - holds **no larger** elements as the remaining 10%.

Can you take any clue from om the above, how the performance of `std::nth_element` roughly plots against the **ratio** of the `n` in its name and the size of the container?

*: For a hint which STL algorithm is applicable here, just continue to read – the answer is near the end of the text.

Partitioning and Sorting Operations (cont.)

Given some STL container, say

```
std::vector<MyClass> data;
```

holding about hundred-thousand `MyClass` objects …

```
class MyClass {
    … //
public:
    int getPriority() const;
    … //
};
```

… what approach is to prefer over a full sort if

- the 25000 objects with the highest priority (as returned from `getPriority()`) need to be processed first
  - though in no particular order, and
- only **after** having completed this it can be decide if the next batch of 25000 needs to be processed too, and
- only **after** having completed this it can be decide if the next batch of 25000 needs to be processed too … (etc.)

Which particular variation seems attractive if the elements …

- … are removed from the container for processing?
- … are still kept in the container after processing?

Partitioning and Sorting Operations (cont.)

Describe in words what the following source code fragment achieves:*

```
using namespace std;
deque<Plank> magazine = { … }; // assume well-filled
forward_list<Plank> dreirad_bed;
double remaining_pathway_length{ … };
auto more_work_to_do = [](double dist) { return (dist >= 0.0); }
while (more_work_to_do(remaining_pathway_length)) {
    const auto from{magazine.end() - min(magazine.size(), 42)};
    nth_element(magazine.begin(), from, magazine.end());
    move(from, magazine.end(), front_inserter(dreirad_bed));
    magazine.erase(from, magazine.end());
    for (auto ¤t_plank : dreirad_bed) {
        remaining_pathway_length -= current_plank.getLength();
        put_onto_muddy_pathway(std::move(current_plank));
        if (not more_work_to_do(remaining_pathway_length)) break;
    }
}
```

*: In a small village lives a farmer who's meadows became muddy from a long period of rain. He wants to reinforce a pathway he often needs to take, so he asks the carpenter to bring some planks. The farmer doesn't know the exact length of the pathway, only that roughly 100 to 150 planks might are required. The carpenter has far more planks in his magazine, differing in length, and not stored sorted. Only each plank has been marked for its length by pencil. Furthermore the number of planks the carpenter is willing to load on his decrepit Tempo Dreirad is limit to 42. How to proceed? (And how to improve?)

Partitioning and Sorting Operations (cont.)

Given an `std::vector<int>`, holding random values between 0 and 999 (total size say 10000 elements, so with many duplicates),

- how much faster is an approach
  - with partitioning only,
  - compared to a full sort
- if you only need to separate the groups
  - 0 … 9 – to be processed first,
  - 10 … 99 – to be processed second, and
  - 100 … 999 – to be processed last,
- with no particular order of elements within any these groups?

Compare the performance your approach against running a full sort.

---

*: As a hint, if you need one: look-up `std::partition`. And as a separate but related part, while you are at it, you may also lookup (or measure) the difference to `std::stable_partition`.

Partitioning and Sorting Operations (cont.)

Compare the speed of `std::is_sorted` to `std::sort`.

- Does the speed of any or both of the above depend on whether the container is already sorted when the operation is started?
- Assuming an already sorted container in both cases, how does the speed relation depend
  - on the container size, and
  - on the element type (say `int` vs. `std::string`)?

Consider the case of a large STL container that needs to be sorted before further use.

- Usually a large initial section is already sorted (from the last time the sort step was applied).
- Only, in the meantime, more entries have been appended to the end.

Given what the STL algorithms provides, what seems to be a good way?[*]

(Also compare the performance of this approach to to running a full sort.)

[*]: As a hint, if you really need one: besides `std::sort`, look-up `std::is_sorted_until`, and `std::inplace_merge`.

Partitioning and Sorting Operations (cont.)

(i) The following is advanced in so far as it expects familiarity with C++11 Multi-Threading, at least to the degree that the use of `std::async` is understood.*

Try to beat the STL implementation of `std::sort` performance-wise:

- Split the work to be done to two equal-sized data blocks.
- For each start an `std::sort` asynchronously,
- then wait for both to end, and
- combine the two sorted blocks with `std::inplace_merge`.

Furthermore (optional):

- Outline the "recipe" to utilize 4 CPU cores, and
- any may be generalise that to 2^N CPU cores.

(With the right approach, even the general case is easy to achieve … much easier as you might thing at first – *but think at first* before you advance to an implementation.)

*: Learning to use `std::async` is not **that** hard, as and you need to know close to nothing about the intricacies of general multi-threading.

Binary Search (on Sorted Ranges)

First guess, then do a practical evaluation of the "Big-O" performance for all four combinations of the following:

Container type:

- `std::forward_list`
- `std::vector`

Element Type:

- `int`
- `std::string`

What performance would you expect, for `std::binary_search`?

What performance would you expect, for `std::find` (linear search)?

From which size a binary search in a container with ForwardIterators starts to beat the linear search in a container with RandomAccessIterators?

Come up with a way to control comparison cost* and demonstrate how – depending on the comparison cost dominance – the "cross-over" point from $O(log_2(N))$ to $O(N)$ shifts in case of a linked list

* Hint, if you need one: the comparison costs for `std::string`-s increase with the length of a common beginning. (But also understand that for the equality comparison shortcuts are possible!)

Set Operations (on Sorted Ranges)

Assuming prepared test data (of sufficient size) in …

```
std::array<int> td; // filled as explained below
```

… compare copying to (1) a binary tree based set, (2) a hash based set, and (3) appending to a vector, sorting, and removing duplicates:

```
/*1*/ std::set<int> s(td.begin(), td.end());
/*2*/ std::unordered_set<int> s(td.begin(), td.end());
/*3*/ std::vector<int> v(td.begin(), td.end());
      std::sort(v.begin(), v.end());
      std::erase(std::unique(v.begin(), v.end()), v.end());
```

Create the test data as random integer values with:*

- no duplicates at all
- 20% duplicates (i.e. 80% unique values)
- 50% duplicates (i.e. 50% unique values)
- 90% duplicates (i.e. 10% unique values)
- all duplicates (i.e. just one single value)

*: A Technique to Create Test Data with that property has been outlined in another micro project.

Set Operations (on Sorted Ranges)

Get familiar with the "Simple Random Test Patter Generator" [SRTPG] and how it is used to analyse the performance of associative containers.

Then adapt (some of) these tests to make use the Set Operations on sorted ranges, provided as STL algorithms.

Is there a measurable difference in performance?

Are there any operations that can be done easily with STL algorithms but not with the container classes or vice versa?

Heap Operations

Get familiar with the "Simple Random Test Patter Generator" [SRTPG] and how it is used to analyse the performance of priority queue adapters.

Then adapt (some of) these tests to use the Heap Operations provided by the STL.

Is there a measurable difference in performance?

Are there any operations that can be done easily with STL algorithms but not with the container adapters or vice versa?

Minimum/Maximum Operations

Given that there is no other "memory" as the sequence to permute:*

- How can `std::next_permutation` determine
  - to return `true` in most any calls,
  - except when "it is through once" completely?
- Same for `std::prev_permutation`? (Only the other way round.)

So, if you came to an answer, explain the following code ...

```
do {
    std::copy(std::begin(data), std::end(data),
              std::ostream_iterator<int>(std::cout));
    std::cout << std::endl;
} while (std::next_permutation(std::begin(data), std::end(data)));
```

... shows 24 permutation here ...

```
int data[] = { 1, 2, 3, 4 };
```

... but fewer here:

```
int data[] = { 0, 8, 1, 5 };
```

Any idea?*

*: Change `std::string` to `const char *` and things might even get more obscure ...

Given the following word-frequency counting example with an automated test …

```
using WordFreq = std::map<std::string, int>;

int main() {
    std::istringstream in{"one two one two three one"};
    WordFreq wfreq;
    std::string w;
    while (in >> w)
        ++wfreq[w];
    std::vector<WordFreq::value_type> result;
    std::copy(wfreq.cbegin(), wfreq.cend(),
              std::back_inserter(result));
    const std::vector<WordFreq::value_type> expected = {
                                            { "one", 3 },
                                            {"three", 1},
                                            {"two", 2} };
    assert(result == expected);
}
```

… why does the test (probably) fail after changing the word container to a hash-based map and which STL algorithm comes to help?

Minimum/Maximum Operations

How can the following program (potentially) be improved:

```
std::list<std::vector<int>> data;
… // assume data has been filled
std::list<int> minimums, maximums;
for (const auto &e : data) {
    minimums.push_back(*std::min_element(begin(e), end(e)));
    maximums.push_back(*std::max_element(begin(e), end(e)));
}
```

First guess which of the following might improve most, then try (and compare):

- Use the `std::transform` algorithm instead of a range-`for` loop to fill the `minimums` and `maximums` container.
- Replace the calls to `std::min_element` and `std::max_element` by a single one to [`minmax_element`].
- Change the `minimums` and `maximums` container to `std::vector`.*
- Change the `data` container to `std::vector` or `std::array`.

*: If you do so, also consider to resize the containers appropriately or reserve space before filling.

Numeric Operations

How could `std::accumulate` help to find in …

```
std::vector<unsigned long long> datawords;
```

… which bits are …

- … `0` in any datawords?
- … `1` in any datawords?

… or which are …

- … `0` in all datawords?
- … `1` in all datawords?

Why might it – depending on the data – be a good idea to prefer a range-`for` loop over `std::transform`?

*: As a hint, think of a short-cut for early termination if the resulting data word is "all `0`-s" or "all `1`-s".

Numeric Operations

Given the following list of moves (into x and y direction) on a planar surface …

```
using std::tuple<long, long> XY;
std::vector<XY> distances;
```

… how to calculate the absolute positions* with an STL algorithm?

```
std::vector<XY> positions;
  // vvv------------------------------ TODO: select algorithm
std::…?…(distances.cbegin(),
         distances.cend(),
         std::back_inserter(positions),
         [](const XY &e1, const XY &e2) {
             return XY{ … };
                    // ^^^------------- TODO: specify calculation
         });
```

And how to reverse the above, i.e. get back from (absolute) `positions` to (relative) `distances`?

C-Library

Why does with …

```
std::array<int, 1000> testdata;
std::generate_n(testdata.begin(), testdata.size(), std::rand);
```

… the following code (correctly) sorts `testdata` in descending order …

```
bool gt(int x, int y) {
    return (x > y);
}
std::sort(testdata.begin(), testdata.end(), gt);
assert(std::is_sorted(testdata.begin(), testdata.end(), gt))
```

… while its C-equivalent fails (in more or less obvious ways)?

```
bool gtc(const void *xp, const void *yp) {
    return (*(const int*)xp > *(const int*)yp);
}
std::qsort(testdata.data(), testdata.size(), sizeof (int), gtc);
assert(std::is_sorted(testdata.begin(), testdata.end(), gtc))
```

After you identified the problem, correct it, and compare performance.

C-Library (cont.)

Determine the differences in performance …

… for the STL Algorithms …

- `std::sort`
- `std::binary_search`

… vs. their C equivalents:

- `std::qsort`
- `std::bsearch`

In case of a really big performance gap, can you clarify the reason?

E.g. might it be

- inlining[*] the comparison
- vs. calling a function?

Or

- two-way comparison (STL)
- vs. three-way comparison (C)

[*]: Note that modern compilers often decide themselves whether to inline a function body in place of a subroutine call, possibly depending on the optimization level or other compiler specific options. **Especially non-`inline` functions might be inlined** though that can be usually defeated by placing *the not-to-be-inlined* function in a separate compilation unit.

Permute Words in a Sentence

Difficulty

advanced: 😟 😠 😐 😉

Requirements

TBD

Steps To Solution

TBD

A Basic `nodups` Algorithm

Difficulty

simple: 😫 😉 😉 😉

Example

From the following input …

```
beware the Jabberwock my son
beware the Jubjub bird and shun
the frumious Bandersnatch
```

… produce that output:

```
beware the Jabberwock my son
Jubjub bird and shun
frumious Bandersnatch
```

Requirements

Understand the general structure of a (templated) algorithm:

- Usually the first two arguments are iterators of the same type, i.e. anything that has
  - an operation for dereferencing,
  - an operation for advancing,
  - may be compared for equal, and
- actually will compare equal the first is increment often enough.

If the result is to be left in another container (like `nodups` should do) * there is another iterator argument that is dereferenced and assigned to.

Furthermore many algorithms (also the one for the task at hand) can use their iterator arguments according to the rules of InputIterators (first argument here) or OutputIterators (third argument here).

Also understand that some kind of memory is required storing the words already seen.

(For a final hint see text after the initial fragment on the next page.)

Steps To Solution

Start with filling the following fragment:

```
template<typename InIter, typename OutIter>
void nodups_copy(InIter from, InIter upto, OutIter dest) {
    … // TBD
}
```

If you find you need the derefenced iterator type at some point, you may use the following:

```
// "throw in" the iterator type --vvv
  … typename std::iterator_traits< … >::value_type …
//  ^^^--- "get back" its dereferenced type ---^^^
```

When an initial version is running, test it with as many kinds of iterators as you can think of, including (but not limited to)

- pointers (as iterators for native arrays),
- stream-iterators (for input and output),
- front-/back-insert iterators (for output),
- and – of course – **combinations** thereof.

An Improved `nodups` Algorithm

Difficulty

advanced to challenging: 😟 😣 😐 😉 / 😟 😱 😐 😎

Requirements

Understand the solution of the Basic `nodups` Algorithm micro-project.

Steps To Solution

- Instead of running a loop inside the algorithm,
- delegate work to an appropriate STL algorithm.*

Also, a lambda with capturing by reference might come in handy.

*: As a hint, if you need one: `std::copy_if` is your friend ... (and if not available because your standard library hasn't been updated from C++98 to C++11 it's also doable with `std::remove_copy_if`).

Extension Centric STL Micro-Projects

1. Implement a *Basic Sequential Map* Container
2. Implement an *Advanced Sequential Map* Container
3. Implement a Polymorphic Container *with* Inheritance
4. Implement a Polymorphic Container *without* Inheritance
5. Implement an "Iterator-Conscious" Container

A *Basic Sequential Map* Container

Difficulty

advanced: 😟 😠 😐 😉

Requirements

TBD

Steps To Solution

TBD

An *Advanced Sequential Map* Container

Difficulty

challenging: 😟 😱 😐 😎

Requirements

TBD

Steps To Solution

TBD

A Polymorphic Container *with* Inheritance

Difficulty

advanced: 😟 😠 😐 😉

Requirements

TBD

Steps To Solution

TBD

A Polymorphic Container *without* Inheritance

Difficulty

challenging: 😟 😱 😐 😎

Requirements

TBD

Steps To Solution

TBD

An "*Iterator Conscious*" Container

Difficulty

challenging: 😟 😱 😐 😎

Requirements

TBD

Steps To Solution

TBD

Miscellaneous Other Micro-Projects

1. Timings of `std::array` Traversals
2. Timings of Container Traversals
3. Timings of Unordered Associative Containers
4. Timings of Various Callback Mechanisms
5. Practical Improvements to Time-Measurements
6. Estimate SBO-Improvements to Memory-Footprint

Timings of `std::array` Traversals

Difficulty

trivial to *simple:* 😉 😉 😉 😉 / 😠 😉 😉 😉

Requirements

TBD

Steps To Solution

TBD

Timings of Container Traversals

Difficulty

simple: 😠 😉 😉 😉

Requirements

TBD

Steps To Solution

TBD

Timings of Unordered Associative Containers

Difficulty

simple to *advanced:* 😫 😉 😉 😉 / 😟 😫 😐 😉

Requirements

TBD

Steps To Solution

TBD

Practical Improvements to Time-Measurements

Difficulty

challenging: 😟 😱 😐 😎

Requirements

TBD

Steps To Solution

TBD

SBO-Improvements to Memory-Footprint

Difficulty

advanced: 😟 😠 😐 😉

Requirements

TBD

Steps To Solution

TBD