C++ STL – Odds and Ends

(Day 2, Optional)

Extensions Provided by Boost

From the libraries available on the Boost Platform, probably the following are the most interesting to augment what is provided in the STL:

- With respect to (additional) containers:
  - Boost.Bimap
  - Boost.MultiIndex
  - Boost.PropertyTree
- With respect to (additional) algorithms:
  - Boost.Algorithm
- With respect to implementing Iterators:
  - Boost.Iterator
- As an alternative to the STL iterator interface for algorithms:
  - Boost.Range

Boost.Bimap

Boost.Bimap semantically* combines two maps so that

- efficient Bidirectional Lookup is provided, or
- as a concrete example
  - look-up IP-Numbers for Host-Names, and
  - llok-up Host-Names for IP-Numbers.

*: The technique used to implement that kind of bidirectional lookup has not been researched for the purpose of that presentation. But it can be assumed – as for most libraries available on the Boost Platform – that is at least as performant as an equivalent "home-grown" solution based on STL components.

Boost.MultiIndex

Boost.MultiIndex is the "in memory" equivalent for data base table, i.e.

- by means of very advanced template programming a versatile parametrized class is provided
- extending the principles of a map to multiple columns,
  - each of which may be
    - indexable (or not)
    - unique (or not)
  - ...

Note that the complexity of advanced template programming is mostly on the implementors side.

> From the client's perspective it rather appears as DSL that can be used in a "Cook-Book" style, following and extending the examples from the documentation.

Boost.PropertyTree

Boost.PropertyTree provides a data structure close to the GoF-Composite design pattern.

It allows for

- hierarchical nesting (in principally unlimited depths)
  - of pairs of
    - keys and
    - values,
  - with the latter introducing recursion by
    - supporting property (sub-) tree
    - besides atomic elements.

> Property trees (from Boost) are especially if it is necessary to make the contained information persistent.

Property Tree Serialisation Formats

There are several serialisation formats to chose from when a property tree is stored to a file or read-back:

- XML
  with pre-defined tags (**not** any generic XML structure);
- JSON
  a hierarchical data format (made popular by JavaScript) for simple data exchange in web applications and elsewhere;
- INFO File
  a hierarchical format (much similar to the former), which was specially designed for storing a property tree in a text file;
- INI-Files
  as in early versions of MS-Windows.*

*: When storing to and reading from ini-files there are limitations to the property tree content, especially ini-files were not designed to hold (and easily edit) nested data structures.

Boost.Algorithm

Boost.Algorithm extends the Algorithm Dimension of the STL.

E.g. there are algorithms applicable to containers

- for efficiently sub-sequences
- applying comparisons
- ... (and more) ...

Since 2011 with each of the released ISO standards* parts of this library were included.

> Therefore Boost.Algorithm also provides a way to get (some) C++11/C++14 compatibility even if not provided by the compiler vendor.

*: I.e. at the time of writing this, with C++11/14.

Boost.Iterator

Boost.Iterator is two things:

1. It is a clean-up (beyond a pure re-wording) for the Iterator Categories as originally specified with C++98.
2. It helps to avoid much systematic code when implementing new iterators.

Boost.Range

Boost.Range implements all STL algorithms with a different interface:

- Instead of supplying two iterators to specify a range by the element it starts with and the first element beyond its end,
- **a single argument** has to be specified
  - which is a **model of some a range concept**, and
  - can be created in various ways.

Of course, creating a range from two iterators is still possible – but it is not any more the **only** way, as it is with the classic STL API.

> Even more flexibility is achieved with respect to ranges by utility operations, e.g. to directly feed the output from one algorithm as input to another one.*

*: As for this operation also an overload is defined for `operator|`, those who are accustomed to U*ix Pipelines* might be appealed also by the nifty look!

STL Concurrency Support

Though C++11 introduced substantial Concurrency Support in the language and library

- the STL **does not provide a direct interface** to distribute work among several threads, as a means to improve performance on multi-core CPUs, instead
- the STL specification gives leeway to exploit chances for parallelizing independent parts of the work,

as long as it stays within the limits of the specified algorithmic complexity.

> **[!]** The problem is not that STL algorithms cannot be parallelized – it is rather that developers might not be aware that it does.

Parallelizing Transparently

E.g. it is **unspecified** by the ISO standard

- in which order `std::count_if` accesses the sequence to process
- which algorithm is used by `std::sort`

Therefore a conforming implementation could well be split the work over a number threads, given its complexity is O(N) or $O(N \times \log_2(N))$ respectively.

> If not explicitly specified for an algorithm, **no particular order** should be assumed in which the elements on a container are accessed.

As this may not have been the case in the past,*

- the problem might occur without warning,
- when old code is compiled with a new library version,
- that implements parallelized algorithms without giving notice.

*: The author confesses guilty for having used at least one example to demonstrate the use of STL algorithms, that had a stateful functor, hence depending on linear sequential access of container elements, front to back, but the algorithm used in the example didn't guarantee this!

Parallelizing Explicitly

If a given library implementation does not exploit chances for parallelizing, it may be done explicitly.

Given there are four CPU cores, sorting a large* array might be done by

- **first** sorting four independent, equally sized segments in four threads – using `std::sort` and a bit of arithmetic with respect to the borders,
- **then** merging the two bottom and the two top segments in two threads – using `std::inplace_merge` (and some recycled border calculation from the first step), and
- **finally** merging the two sorted sections resulting from the previous step – again using `std::inplace_merge` on the whole array.

[!] Though the above sounds simple – and in fact is not that hard to achieve with what is already provided by the STL – understand that any added complexity tends to make a program harder to understand and more difficult to maintain.

Improving Performance - A Word to the Wise

Before applying a performance improvement, be sure to understand the following:

- If some part of a complex task contributes 20% to its total runtime,*
  - a performance improvement of 30% wins 6% total,
  - a performance improvement of 50% wins 10% total, and
  - **no** performance improvement will ever win more than 20%.
- Other if some part of a complex task contributes 90% to its total runtime,
  - then … (you will surely be able to do the math yourself).

> The total effect of performance improvements often are hard to predict.

Therefore:

- **Measure before you start** …

… and be sure to measure correctly and check the results for plausibility!

*: This may or may not be much: if applied to a large application, **most** of its whole source code may not have that impact. When considering an algorithm in isolation it may not be much if it is the core of the algorithm, not some preparation, book-keeping, or handling of exceptional conditions.

Scalability to Multiple Cores

The following is well-known – and can be demonstrated by measurements:

> To increase performance it is contra-productive to split a CPU-bound task to more threads as there are physical cores.*

When parallelizing with [Packed Tasks] in many cases

- the decision in many cases is best left to the library (implementation),
- assuming it has platform or operating system specific means to adapt actual (hardware) concurrency in the optimal way.

> **[!]** **Explicitly exempt** from that advice are more or less sophisticated networks of **tasks that run independently** from each other, **communicating via buffered pipelines**.

Such designs easily deadlock if the library is allowed to silently turn asynchronous into synchronous calls.

---

*: Or at least to more threads as there independent partial hardware processing units that can work physically concurrent, like there are in hyper-threading architectures.

STL-Quirks and Shortcomings

Though the STL has been designed by only two original contributors and therefore has a quite uniform look&feel, there is some "historic ballast" that is difficult to get rid of.

The areas considered on the following pages are:

- A few case Inconsistent naming.
- Some non-obvious Peculiarities of "Big-O" complexity specifications.

Both are considered more closely on the next two pages.*

*: To give credit where credit is due: this whole section was inspired by some sequences from the following Scott Meyers video, starting at minute 37:54: https://www.youtube.com/watch?v=5tg1ONG18H8

STL Naming Inconsistencies

Purging an element with a given value

- from an `std::list` and an `std:forward_list` is done with the member function `remove`, while
- from any associative container it is done with with the member function `erase`.

Sorting elements

- of an `std::list` or an `std::forward_list` needs to be done with the member function `sort`
  - which guarantees a Stable Sort, while
- for any other sequential container it is done with the (non-member) algorithm `std::sort`,
  - which **is not guaranteed to be a stable sort**, and
  - as if this is desired `std::stable_sort` has to be used.

(More cases as those discussed above may exist.)

STL Non-Obvious "Big-O" Complexity

Usually operations are not implemented if this were only possible with a substantial performance penalty.

- Therefore `std::vector` has no member function `push_font` as `std::deque` and `std::list` have ...
  - ... but you can still use the `insert` member function anywhere, though $O(N^2)$ performance.
- Nonetheless `std::binary_search` is applicable to linear lists and specified to have "good" $O(\log_2(N))$ performance ...
  - ... though in this case you would naively expect much worse O(N).
- Adding elements to an `std::vector` at its end is specified to run in amortized constant time,
  - even though the available space must be eventually enlarged,
  - an operation more and more expensive, each time it happens.

(More cases as those discussed above may exist.)