C++ STL – Providing Extensions

(Day 2, Part 2)

- STL-Design — Recapitulation
- Extending the Container Dimension
- Extending the Algorithm Dimension

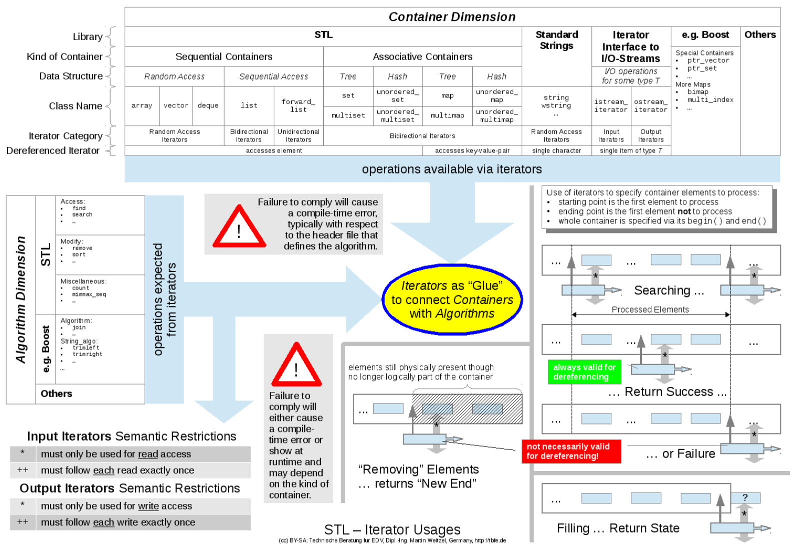
STL-Design — Recapitulation

Basically the STL is designed along two axes

- Containers and
- Algorithms, with
- Iterators providing the glue.

The latter fall into one of several Iterator Categories,

- containers specify the Iterator Category Provided,
- while algorithm specify the Iterator Category Required.



Container Dimension

Library	**STL**									**Standard Strings**	**Iterator Interface to I/O-Streams**		**e.g. Boost**	**Others**
Kind of Container	Sequential Containers					Associative Containers								
Data Structure	*Random Access*			*Sequential Access*		*Tree*	*Hash*	*Tree*	*Hash*		*I/O operations for some type T*		Special Containers: • `ptr_vector` • `ptr_set` • … More Maps: • `bimap` • `multi_index` • …	
Class Name	`array`	`vector`	`deque`	`list`	`forward_list`	`set`	`unordered_set`	`map`	`unordered_map`	`string` `wstring` …	`istream_iterator`	`ostream_iterator`		
						`multiset`	`unordered_multiset`	`multimap`	`unordered_multimap`					
Iterator Category	Random Access Iterators			Bidirectional Iterators	Unidirectional Iterators	Bidirectional Iterators				Random Access Iterators	Input Iterators	Output Iterators		
Dereferenced Iterator	accesses element							accesses key-value-pair		single character	single item of type *T*			

operations available via iterators

Algorithm Dimension	STL	Access: • `find` • `search` • …
		Modify: • `remove` • `sort` • …
		Miscellaneous: • `count` • `mimmax_seq` • …
	e.g. Boost	Algorithm: • `join` • … String_algo: • `trimleft` • `trimright` • … …
	Others	

operations expected from iterators

Failure to comply will cause a compile-time error, typically with respect to the header file that defines the algorithm.

Iterators as “Glue” to connect *Containers* with *Algorithms*

Failure to comply will either cause a compile-time error or show at runtime and may depend on the kind of container.

Input Iterators Semantic Restrictions

*	must only be used for read access
++	must follow each read exactly once

Output Iterators Semantic Restrictions

*	must only be used for write access
++	must follow each write exactly once

Use of iterators to specify container elements to process:
- starting point is the first element to process
- ending point is the first element **not** to process
- whole container is specified via its `begin()` and `end()`

elements still physically present though no longer logically part of the container
“Removing” Elements … returns “New End”
Searching …
Processed Elements
always valid for dereferencing
… Return Success …
not necessarily valid for dereferencing!
… or Failure
Filling … Return State
?
*
…

STL – Iterator Usages

(cc) BY-SA: Technische Beratung für EDV, Dipl.-Ing. Martin Weitzel, Germany, http://tbfe.de

Container Dimension

Besides the containers provided by the STL, there are already various extensions to the container axes:

- Native (C-style) Arrays
- `std::basic_string` with all its convenience aliases
  - `std::string`
  - `std::wstring`
  - `std::u16string`
  - `std::u32string`
- `std::istream` and `std::ostream`
  - via `istream_iterator`
  - and `ostream_iterator`
- Containers provided by Boost
  - Boost.Bimap
  - Boost.MultiIndex
  - … (and more) …

Algorithm Dimension

Most of the algorithms provided by the STL Algorithm Library and the STL Numeric Library deal with accessing or elements in a container or provide some information with respect to the content of a container.

- Extensions to these have been provided by Boost.Algorithm.*
- Some of this was standardised with C++11.

Algorithms in the C++ standard library **not using** iterators as generic interface to a container are:

- `std::max`, `std::min`, `std::minmax` – defined in `<algorithm>`

Algorithms operating **directly on iterators** are:

- `std::advance`, `distance`, `std::next`, `std::prev` – defined in `<iterator>`

*: Note that what is provided by [Boost.String_algo] are of course algorithms in the meaning of a (library) subroutine that carries out a certain operation, but these algorithms are **not decoupled via iterators** from the string representation (understood as "character container").

Iterators as "Glue"

Decoupling algorithms from containers via iterators, abstracting the operations for

- accessing a container element, and
- advancing to another element

is a key attribution Alexander Stepanov and David Musser made to the direction into which C++11 evolved since it was first standardised by ISO and ANSI as C++98.

Contemporary class libraries at that time rather

- **used C++ templates** to generalise on the container element type, but
- **not with algorithms** to decouple element access and repositioning inside the container via iterators.

Iterator Categories

Iterators are usually implemented as (container specific) classes, but there is one important thing to understand with respect understand about iterator categories:

Iterator Categories **do not form a classic class hierarchy** and have no inheritance relationship to each other.

This is true despite the fact that frequently the operations of one category are extended by another category.

Rather iterator categories should be understood as contracts.[*]

(i) For more information on categories and especially how some category relates to other categories see: http://en.cppreference.com/w/cpp/iterator

[*]: With respect to iterator categories also another terminology is common, which uses the key abstractions of "*concepts*" *and* "*models*". That terminology is applicable on a broader base and uses the word "*model*" to describe a set of requirements, that need to be fulfilled by a class which is said to be "*a model of the concept*".

Iterator Operations

The obvious aspect from an iterator category is which operations are provided.

Any violation of that part of the contract causes a diagnostic at compile time and translation will usually fail.*

What is left as *Quality of Implementation Issue* (or QoI-Issue) is how the diagnostic points out what went wrong – which in principle is not an easy decision for the compiler as there are two parties involved:

- The client code (using the STL) which probably did adhere to the contract with respect to the required operations?
- The STL code (which implements an algorithm) which might have used an operation not part of the specified iterator category?

Therefore typical diagnostics show a lot of context information (meant to be helpful to decide who is the culprit, but often rather distracting)

*: Since C89 the ISO/ANSI are not explicit about the conditions for which an executable is the end result … for the simple reason that interpretative translation environments should not be out-ruled.

Iterator Semantics

The more critical part compared to operations are certain semantics expected from an iterator category.

> [!] This type of violation may result in failure to work as expected, or even even be the cause for Undefined Behaviour.

Examples for that sort of the obligations contracted via the iterator category are:

- **Multi-Pass Capability:**
  It may or may not be acceptable to store the start-point iterator and use it again, to pass over the same sequence a second time.
- **Input and Output Sequences:**
  A conforming client is required to use access (`operator*()`) and advance (`operator++()`) alternately.

> The crucial part here is that with too little testing
> any problems may go completely undetected.

Iterator Invalidation

It should also be noted that certain operations on an

- `std::vector`

may cause Iterator Invalidation.[*]

The same is true for

- `std::basic_string`

with all its usual type aliases, like `std::string`, `std::wstring` etc.

[!] Any further use of the invalidated iterator(s) – except destruction – may cause Undefined Behaviour.

[*]: It is probably self-evident that for also any iterator denoting a container element that gets removed from the container will be invalidated – or at least+ will not point any longer to the element it once pointed to.

Iterator Category Provided

A container usually specifies which Iterator Category it **provides**.

This is not less and not more as a shorthand for the contract

- the iterators of a given container are able to fulfill
    - talking in terms of provided operations, and also
- the usage restrictions they depend on.

From the STL standard containers

- all provide the capabilities of the BidirectionalIterator category,
- beyond that `std::array`, `std::deque`, and `std::vector` even provide the capabilities of RandomAccessIterator category.

Iterator Category Required

An algorithm usually specifies which Iterator Category it **requires**.

This is not less and not more as a shorthand for the contract

- the iterators as used in the algorithm needs to fulfil
  - talking in terms of provided operations, with violations usually detectable at compile time, and also
- what the algorithm promises to oblige to
  - talking in terms of usage restrictions, typically not detectable at compile time.

For the STL algorithms details vary between

- iterators specifying an input range
- and those specifying an output sink.

Iterator Category Required for Input

Most STL algorithms[*] access a container – as a sequence of values – by requiring two InputIterators to specify the range to process.

Two things are to be noted here:

1. InputIterators do **not provide the multi-pass capability** (which is part of ForwardIterators).
   Hence most algorithms can directly process a stream input using an `std::istream_iterator`.

2. The STL specification for some algorithms provides leeway to split the work among several threads, so that the processed sequence **may or may not** be sequentially accessed.

More technically: An implementation of an algorithm may apply techniques usually known as *Compile Time Meta Programming* to change its internals depending on the actual iterator category it receives.

[*]: With the notable exception of `std::sort`, which requires [RandomAccessIterators], but **not** `std::binary_search`, which requires ForwardIterators (besides a sorted range to search, of course).

Iterator Category Required for Output

STL algorithms with a specified output sink (typically the `_copy`-variant of some algorithm family) require OutputIterators.

Hence these algorithms (also) can directly generate stream output using an `std::ostream_iterator`.

If you implement algorithms with an output sink yourself, be sure to thoroughly test with different kind of iterators to write the sink.

E.g. it is not unusual absolutely correct for a model of [OutputIteraror] to put all the work in `operator*()`, ignoring `operator++()` completely.*

[!] Therefore any errors with respect to the correct alternating use of dereferencing and advancing may go undetected, when testing exclusively with these kind of iterator implementation, but will fail when used with a native pointers.

*: More precisely, `operator*()` fill return a special surrogate type for which `operator=()` is overloaded to do all the work.

Extending the Container Dimension

Extending the STL at the container dimension basically means that a new container class should offer an iterator interface.

These are the basics (given the new container is named `MyContainer` templated in its element type symbolically denoted as `T`):

- There should be a nested class named `MyContainer<T>::iterator`.
- It should provide (at least)*
  - `T& operation*()`
  - `MyContainer<T>::iterator operator++()`
  - comparisons for equals and unequals
- Furthermore the container itself should have member functions `begin()` and `end()` returning `MyContainer<T>::iterator`.

*: Of course much more is desirable, like `const_iterator` nested class, with matching `cbegin()` and `cend()` member functions, and more operations, depending on the Iterator Category that makes sense for the kind of container and is easily supported by its implementation.

Pragmatic Container Iterators

The pragmatic way to provide iterators for a new container is as follows:

- Just provide the minimum, and
- depending on the use cases of the container iterators,
- by and by add everything that is missing, as the need occurs.

The following shown an absolute minimum bare-bones fragment that might can serve as a starting point:

```
template<typename T>
class MyClass {
    …
public:
    …
    struct iterator {
        … //
        T& operator*() const { … }
        iterator operator++() { … }
    };
    iterator begin() { … }
    iterator end() { … }
};
```

Fully Compliant Container Iterators

Writing a "fully compliant"

- `MyClass::iterator` (that e.g. supports pre- and post-increment),
- a corresponding `MyClass::const_iterator`
- with `cbegin()` and `cend()` member functions in `MyClass`,
- maybe variants iterating in reverse too
- … etc. etc. …

quickly turns into writing much and partially very systematic, but nonetheless unavoidable code.*

Especially there are some requirements in correctly "tagging" Iterators depending on their category, that must be followed.

(i) For more information on the requirements of the various iterator categories start with reading section Iterator Primitives in: http://en.cppreference.com/w/cpp/iterator

*: Boost.Iterator provides some support with respect to the boiler-plate parts.

Extending the Algorithm Dimension

In comparison, extending the algorithm dimension is easier by far.

- Assume as little as possible about the container(s) your algorithm should work on.
- Consequently and consistently use iterators of an appropriate category to access a container content.
- May be get comfortable with
  - the basic techniques of Compile Time Meta Programming, or
  - as the very least understand where `std::iterator_traits` comes in handy, i.e. which kind of problems it will easily solve.

Furthermore, if there is the option,

- instead of implementing new algorithms from ground-up,
- prefer to add new algorithms by reuse.

New Algorithms from Ground-Up

As an example how to implement a (not yet existing) algorithm from ground-up, consider `std::copy_if`, which was missing from C++98 and has been added in C++11.

Basically, implementing `copy_if` from ground-up could look as follows (in `namespace std`):

```
template<typename InIt, typename OutIt, typename Pred>
OutIt copy_if(InIt from, InIt upto, OutIt dest, Pred tocopy) {
    while (from != upto) {
        const auto temp{*from++};
        if (tocopy(temp))
            *dest++ = temp;
    }
    return dest;
}
```

For the local variable, prior to C++11 the following had to be used:

```
const typename std::iterator_traits<InIt>::value_type temp;
```

New Algorithms by Reuse

Adding `std::copy_if` is also possible by reusing the existing algorithm `std::remove_copy_if`.

Then the implementation comes down to (again in `namespace std`):*

```
template<typename InIt, typename OutIt, typename Pred>
OutIt copy_if(InIt from, InIt upto, OutIt dest, Pred tocopy) {
    using VType = typename std::iterator_traits<InIt>::value_type;
    return std::remove_copy_if(from, upto, dest,
                               [&tocopy](const VType &e) {
                                   return !tocopy(e);
                               });
}
```

With C++14 `VType` in the lambda argument list can be replaced with `auto` (and hence the whole type alias definition for `VType` be dropped).

*: Some may prefer the function adapter `std::not` over the lambda, turning the body of the above into:

```
return std::remove_copy_if(from, upto, dest, std::not1(tocopy));
```

Practically Extending the STL

For a practical approach to extend the STL visit the Mini Projects focussing on extending the STL.

(i) For more information on the STL Algorithm Library see:
http://en.cppreference.com/w/cpp/algorithm