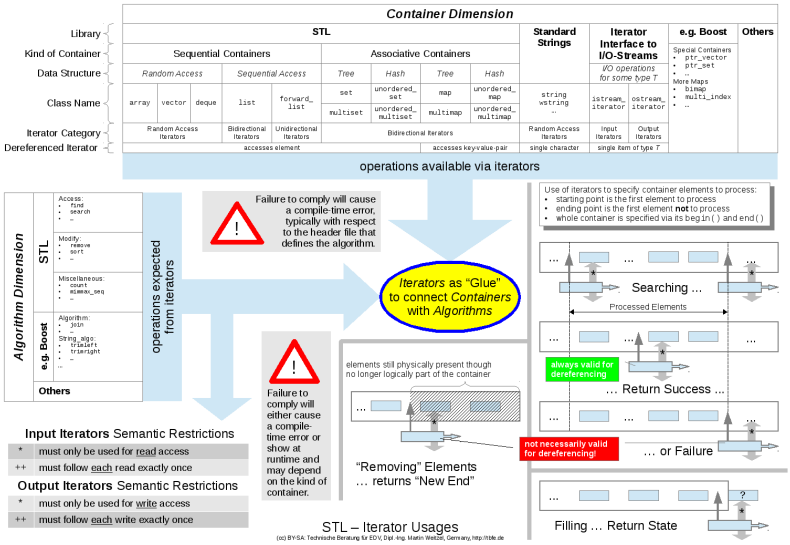
C++ STL – Fundamental Architecture

(Day 1, Part 1)

Architectural Overview

- Container Dimension
- Algorithm Dimension
- The Role of Iterators



The Container Dimension

Along the Container Dimension there are*

- 10 Standard Container Classes, of which
  - (only) 9 are considered part of the STL in the narrower sense, but
  - as the standard string class provided all the typical member functions of containers (with sequential and random access) it may be counted here too;
- furthermore 3 Container Adaptors (creating containers with special features or restrictions based on the above).

> (i) For more information on STL container classes see:
> http://en.cppreference.com/w/cpp/container
> http://en.cppreference.com/w/cpp/string

*: The classic containers are defined in the header files `<array>`, `<deque>`, `<forward_list>`, `<list>`, `<map>`, `<set>`, `<unordered_map>`, and `<unordered_set>`; container adaptors are defined in header files `<queue>`, `<stack>`, and `<priority_queue>`; the standard string class is defined in header file `<string>`.

The Algorithm Dimension

At the Algorithm Dimension there are (counting what is provided by the header file `<algorithm>`)

- (about) 80+ entries, from
- (about) 30 related groups (or families*),

which may be considered augmented by the header ` contributing

- another 5 algorithms.

> (i) For more information on STL algorithms see: http://en.cppreference.com/w/cpp/algorithm and subsection Generic Numeric Operations in http://en.cppreference.com/w/cpp/numeric

*: There are no strict rules about grouping algorithms to families. It could be done by name prefix – e.g. by putting `copy` in the same group as `copy_if` and `unique` in the same group as `unique_copy`, or by similarities in purpose – e.g. by putting `all_of`, `none_of`, and `any_of` in the same group, or everything that has to do with permuting elements, everything that has to do with sorting ...

Role of Iterators

Viewed from the big perspective,

- STL Iterators just "provide the glue" that
- brings together Containers and Algorithms,
- without having the one to know details of the other.*

The latter is a big advantage for making the architecture extensible.

*: At the time the STL was designed by Alexander Stepanov and David Musser this was a rather uncommon and maybe even unique approach. Contemporary C++ class libraries either imitated the style used in other OO-languages, that had a common base class for all kinds of objects, or – after templates came into more widespread use – implemented a more or less closed set of standard containers all complete with member function implementations for the expected use cases.

Abstract Specification

One of the most distinguishing marks of the STL its strong preference for abstract specifications.

Originally and formally until today, the STL documentation disclosed few abouts its implementation,* nor with respect to data structures, neither for details on the algorithms used.

Instead:

- most often just performance guarantees are defined (which of course give clues or even certainty with respect to the implementation);
- for iterators a number of categories are defined, from which the interplay of containers with algorithms – including the restrictions – can be derived.

*: Emphasis is to put on "documentation" here, as the STL, being to nearly 100% a "header file only" library, could effectively hide about its implementation.

Consequences of Abstract Specifications

Sticking to abstract specifications has an important impact on writing correct code:

A substantial number of software developers tends to save the pains of studying the documentation for libraries they use.

Instead, a "trial and error" approach is adopted:

- A piece of code is considered to be "correct"
  - if it compiles without an error message*, and
  - passes some simple – far from exhaustive – testing.

[!] Completely ignoring the written documentation of the STL can lead to (compile time) error message hard to decipher or even undefined behaviour (at run time).

*: Furthermore and increasingly, there is the expectation the compiler should give a concise description of what is wrong (as if it could read the intentions of the developer) and ideally describe the appropriate fix in the message.

Breaches of Contract

With respect to the STL there are two substantially different ways to breach the contract and use the components in unsupported ways:

1. At compile time:
   - Currently only sometimes a concise error description is the result.
   - More often it causes myriads of messages, difficult to decipher.*
2. At run time:
   - In this case – at least principally – "all bets are off", but
   - quality implementation sometimes add checking code during *Debug Builds* or may allow to enable selected STL safety checks for *Release Builds*.

> **[!]** Using the STL with implementation dependent additional checks enabled may incur both: (Much) **slower execution** and (much) **larger memory footprint.**

*: At least for the "un-enlightened" ... (though the situation may improve once Concepts Lite are part of the language, which is expected for C++1z).

Compile Time Errors with STL Components

Compile time errors usually result from an insufficient understanding which set of operations a class must be provide to be acceptable for the instantiation of some template.*

- The typical result are compiler error messages with a whole bunch of information
  - how the situation was reached, often including,
  - every alternative considered to resolve the problem, and
  - why finally everything failed.
- with the often only important piece buried somewhere inside.

Which line of source code triggered the whole drama?*

*: Besides reading the documentation, until sufficient experience is acquired with respect to sorting out the irrelevant from the relevant pieces in STL error messages, the usual advice is to add lines to your code in baby steps, i.e. intersperse your coding with frequent compilation (syntax checks usually suffice, no need for a complete build) so that it is immediately obvious which change you made triggered the problem.

Undefined Behaviour of STL Components

To allow – **not to mandate(!)** – highly efficient implementations, the C++ standard often imposes little obligations on run time error checking.

> **[!]** Breaking documented pre-conditions that cannot be checked at compile time may – **but not necessarily needs to(!)** – lead to catastrophic failure at run time.

Before complaining over an inflationary use of the term Undefined Behaviour in the C++ standard, be sure to understand the following:

- C++ has a very broad user base with widely varying demands.
- **Undefined behavior provides the leeway to allow any implementation to meet the specific needs of its user base**.
- Therefore any C++ implementation is at liberty to replace undefined behaviour with defined behaviour.*

*: Obviously this assumes such needs are sufficiently common and do not cause too many users to migrate away, who for which the price is not acceptable (typically some loss of performance or an enlarged memory footprint).

Concepts and Models

Two key terms in the STL documentation are "Concept" and "Model".*

A concept defines a (usually templated) type by

- use patterns that need to be valid, e.g. required
  - embedded type definitions,
  - member function, or
  - acceptance by overloads.

A model states for some type that it

- actually fulfills a concept*, hence making that type acceptable
  - to instantiate a templated type,
  - which requires that concept.

For a concept *C* and a model *M* the wording often is: *M* is a model of *C*.

---

*: Concepts and models are related to what is provided by an *interface* and its *implementations* at run time, but apply at compile time, much in the way of Duck Typing. For a video lecture by Bjarne Stroustrup how concept and models may be made visible to the compiler (and hence get out of their "documentation only" role), see: http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/A-Concept-Design-for-C-

Common Data Structures

The term data structure usually means to somehow combine several pieces of information so that they can be handled as a unit.

Despite the many variations there are only two main principles:

- To get from one piece to another piece
  - either requires some address calculation with respect to the first (or the data structure in its entirety),
  - or following a pointer being part of the first (or used to identify the whole unit).
- Hence a data structure may be stored
  - in closely adjacent memory locations, or
  - more or less spread-out over all memory
  - (or a mix of both).

Adjacent Memory

Typical for adjacent memory is to store related pieces of information in

- a `struct` (C or C++)
- a `class`* (C++)
- a native array (C and C++),
- an `std::array` wrapper (C++),

Furthermore (after the first element is located) also

- `std::vector` and
- `std::basic_string`

store adjacent elements (chars) in adjacent memory, and

- `std::deque`

stores splits the whole data in smaller chunks, each of which is stored in adjacent memory.

---

*: In C++ a `class` and a `struct` are basically the same, only that the first starts with private and the latter with public members (data and functions). Once the first `private:` or `public:` label has been explicitly inserted, all difference is gone.

Linked Lists

Typical for spreading out data structures over memory is to store related pieces of information in

- `std::list` and
- `std::forward_list`.

Each element (i.e. piece of information) stores the memory address of its neighbour(s).

Therefore the per-element overhead is one (`std::forward_list`) or two (`std::list`) pointers (in addition to possible padding).

> Searching through lists requires to visit any element, one after the other, until the element of interest is found.

Binary Trees

Binary tree structures also spread out data structures over memory..[]

All the associative containers from C++98, i.e.

- `std::set`,
- `std::multi_set`
- `std::map`, and
- `std::multi_map`

are stored in some form of binary tree, called Red-Black-Tree.

The per-element overhead is two pointers, as – besides the element it holds – each "node" points to two sub-trees, usually called "left" and "right".

> The left sub-tree consists of nodes only with elements* comparing *less-than* to itself, and the right sub-tree holds the others, searching through a binary tree is considerably faster as searching through a linked list.

*: Only for sets the element value is compared, for maps it is the key value.

Hash-Tables

The unordered counterparts to sets and maps, added in C++11, i.e.

- `std::unordered_set`,
- `std::unordered_multiset`,
- `std::unordered_map`, and
- `std::unordered_multimap`

are based on Hash-Tables and therefore are a mix of "adjacent" (for looking up keys) with a tendency to "spread-out" (in case of overruns).

> Sequentially accessing any of these will result in poor cache performance on modern hardware architectures.

Hash-Tables Explained (1)

The idea of hash tables starts with having a Hash-Function that calculates a numeric index from an element's value (for sets) or key (for maps).

E.g. for a classic `char` string this may be done as follows:

```
std::unsigned str_hash(const char *s) {
    constexpr int p[] = { 3, 5, 7, 11, 13 };
    constexpr int N = sizeof N / sizeof N[0];
    std::size_t r = 0, i = 0;
    while (*s) r += (0xFF & *s++) * p[i++ % N];
    return r;
}
```

For the actual storage adjacent memory is chosen, usually 10 ... 30% oversized, and the result of the hash function is then folded into that range of values.*

*: This could be done in many ways, with a modulo operation being the simplest, but – depending on the hash function – not necessarily the best for an equal spreading over the whole range, with the same probability for all outcomes.

Hash-Tables Explained (2)

In that way, i.e. assuming

```
constexpr HASH_TABLE_SIZE = 1200; // expecting ~1000 elements
const char *data[HASH_TABLE_SIZE];
const char* element;
…
```

the index where an element is to be stored can be quickly determined

```
data[str_hash(element) % HASH_TABLE_SIZE] = element;
```

or the existence of an element may be looked-up:

```
if (data[str_hash(element) % HASH_TABLE_SIZE] != nullptr) …
```

Hash-Tables Explained (3)

What is still left is to solve is the problem of over-runners, i.e.

- different elements hashing to same value or maybe
- hashing differently but being folded to the same index.

There are two classic strategies:*

1. **Elements are not directly stored in the hash table.**
   Instead, each entry – also called hash-bucket – is the start of a linked list, holding all elements assigned to that entry.

2. **Neighbouring free cells are used.**
   This solution is good under the aspect of proximity (see next page), but has implications for search times of all entries,i.e. the non-overrunning too.

---

*: With only some few degenerated hash function results, giving the same result for some groups of elements, with the *linked-list solution* the performance degradation is a local problem only (affects members of these groups only). With the *use neighbouring cells* solution it might contaminate large sections of the table and can get especially troublesome if not only new elements are inserted into a hash-table but existing ones need to be removed.

Caching Considerations

Improving efficiency through cached access to main memory has become increasingly important with modern CPU designs.

> Adjacent storage is crucial for achieving good hit-rates in the several cache levels usually preceding slow main memory.*

The overall effects of caching are often are not (yet) reflected in older literature and more theoretical performance considerations.

- This does not mean that everything written ten or more years ago is not applicable any more today (or even plainly wrong).
- It only means that the enormous coverage the performance of data structures has received in information technology should be read with the proverbially "grain of salt".

Usually it will be the best to test and evaluate the performance of the actual code, based on realistic use patterns.

---

*: For a video lecture by *Chandler Carruth* expanding on this see (from minute 34:47) in: https://www.youtube.com/watch?v=fHNmRkzxHWs

Container Sub-Categories

Technically the STL groups its containers into the sub-categories of

- Sequence Containers,
- Associative Containers, and
- Unordered Associative Containers.

The difference between the first and the other two is:

- Sequence containers have a *front* and a *back*, i.e. two distinct "sides" to which insertions may refer elements,*
- while both flavours of associative containers decide on their own where new elements are placed.

For sequential access to all elements the order *begin* (front) to *end* (back)

- depends on the insertion order for the first sub-category.
- is in ascending sort order of the elements value or key for the second,
- in an unspecified order for the third.

---

*: Note that nevertheless an `std::vector` restricts adding (and removing) elements efficiently to its *back*-side (no `push_front` or `pop_front`) while an `std::forward_list` restricts this to its *front*-side (no `push_back` or `pop_back`).

Sequential Access

Sequential access is supported by all containers.

As of C++98 the typical sequential access looked like follows, with `Container` replaced by a concrete instantiation of any STL container, or a type alias):*

```
Container c;
for (Container::iterator it = c.begin(); it != c.end() ++it) …
```

To ensure non-modifying access in case of a modifiable container, the loop iterator must be of type `…::const_iterator`.

---

*: With C++11 some variants became possible.

```
// with auto-typed iterator:
for (auto it = c.begin(); it != c.end() ++it) …
for (auto it = c.cbegin(); it != c.cend() ++it) … // non-modifying
for (auto it = std::begin(c); it != std::end(c) ++it) …
for (auto it = std::cbegin(c); it != std::cend(c) ++it) … // non-modifying, C++14 only

// with range-based loops and auto-typed placeholder:
for (auto e : c) … // by-value access (non-modifying)
for (auto &e : c) …
for (const auto &e : c) … // non-modifying
```

Random Access by Index (Offset)

Another frequent type of access is with a numerical index, which is supported by `std::array`, `std::vector`, and `std::deque`.* The index is basically an integral value,

- enumerating the range of elements,
- in good C-tradition with 0-origin.

The following examples return a reference to the element at position `i`:

```
Container c;
std::size_t i;
…           // requires 0 <= i and i < c.size() ...
… c[i] …    // ... may cause undefined behavior otherwise
… c.at(i) … // ... will throw std::out_of_range otherwise
```

*: Efficiency is directly coupled with the simplicity of the required address arithmetic, which is (slightly) more complex for `std::deque` as for both of the others.

Random Access by Key

The operations shown on the previous page are implemented for `std::map` and `std::unordered_map`, where the key is to be used as index.

Assuming a map-container with keys of type `std::string`, the following examples return a reference to the element associated with key `k`:*

```
Container c;
std::string k;
…
… c[k] …    // returns existing OR NEWLY CREATED entry for key k
… c.at(k) … // will throw std::out_of_range for non-existing keys
```

*: Another way to access an element for a given key is to obtain an iterator to it. The non-existence then can be tested by comparison th the end-iterator of the given container.

Understanding Iterators

A client may expect a certain set of operations, which an iterator has to support, e.g.:

- copy construction and assignment,
- comparisons (at least for equal / unequal)
- Increment (at least, often also decrement)
- dereferencing*

> These operations deliberately are provided in the syntax of (equivalent) pointer operations.

As a later chapter covers iterators in much more detail, the following pages are only meant as quick overview.

*: The operations possible after dereferencing may be limited too *value access* or *assigning a new value*. Technically the former works by `const`-qualifying the result, the latter uses a proxy-type as result, for which (only) assignment is implemented.

Iterator Classes

```
class SomeContainer {
    …
public:
    typedef … value_type;
    class iterator;
    …
};
```

Iterators are usually implemented as nested classes of the container to which they belong.

With **many** necessary details omitted, for some container as sketched on the left …

… the iterator class may (roughly) look as sketched below:

```
class SomeContainer::iterator {
    …
public:
    iterator(const iterator &);
    iterator& operator=(const iterator &);
    value_type &operator*() const;
    friend
    bool operator==(const iterator &rhs, const iterator &rhs) {
        return …; // rhs and lhs denote the same container element
    }
    …
};
```

Pointers as Iterators

Especially for containers storing their elements in contiguous memory, plain pointers may suffice as iterators:*

```
class SimpleContainer::iterator {
    …
public:
    typedef … value_type;
    typedef value_type *iterator;
    typedef const value_type *const_iterator;
    iterator begin();
    const iterator begin() const;
    const_iterator cbegin() const;
    iterator end();
    const_iterator end() const;
    const_iterator cend() const;
    …
};
```

*: Note that the examples is only meant to demonstrate the principle. No STL container will actually use this, as the standard has more requirements on their iterators, which could not be fulfilled that way.

Iterators for Native Arrays

The main advantage of using pointer operations for iterators is that it made native (C-style) arrays compatible with STL algorithms.

For an array `arr` with a size of `N`

- a begin iterator may be obtained by taking the address of the first element, i.e. `arr` or `&arr[0]`;*
- an end iterator may be obtained by taking the address beyond the last element, i.e. `arr+N` or `&arr[N]`.

[!] Taking the **address of** `arr[N]` is always possible (and it must compare greater as all addresses from elements of `arr`), but **accessing** that location in memory has undefined behaviour.

*: Note that `arr` and `&arr` denote different types and the latter is **not suitable** for iterating over all elements of `arr`.

Restrictions on Iterators

As the standard does not intend to out-rule implementations combining **high runtime performance** with **low memory footprint**, some requirements on iterators are deliberately chosen very loose.

> **[!]** Programmers must be aware of this or otherwise risk undefined behaviour of the resulting code.*

Usually it is possible to implement iterators for STL containers

- with the same memory footprint as native pointers,
- their operation dereferencing with the same runtime performance as native pointers, and
- their increment operation comparable to a native pointer increment or assignment.

---

*: One of the most important drawbacks of this is that bad effects may not show in test scenarios or depend on data constellation difficult to reproduce.

Valid Iterator Positions

Iterators may refer to

- any element in an STL container
- **plus one more valid position**.

The latter is usually understood as "one beyond the last", but implementations are only required to behave "as if" in this respect*

> **[!]** Dereferencing an iterator has undefined behavior if it does not point to an existing element of a container.

*: Especially, while "one beyond the last" is easily defined for contiguous memory, it is much mire difficult to come ob with a definition for linked list or stream iterators.

Comparing Iterators (Reachability)

For most containers iterators can only be compare for equal (==) or unequal (!=).

In general there is no easy way to decide whether some iterator incremented a number of times will compare equal to another iterator before it reach the end of the container before.*

Therefore, a typical requirement of algorithms is that an **iterator is reachable** by some other iterator, in the following meaning:

- When applying increment operations, the latter will stay valid
- until it finally compares equal to the former.

[!] Programmers need to be aware of this and strictly ensure the reachability requirement. Otherwise iterators may assume invalid position and access memory outside the container.

*: As always with behavior that is undefined per C++ standard, implementations may chose to catch errors. Especially for iterators that provide a relational comparison a "not-reachable" error might be caught internally.

Invalidating Iterators

For most – **but not all** – containers their iterators stay valid (i.e. always denote the same element)

- once the iterator has been set to an existing element and
- as long as that is not removed from the container.

Notable exceptions from this are containers storing their elements in contiguous memory, when the allocated storage is exhausted and must be extended.

As heap storage often cannot be extended in place, its content may need to be moved in memory, thus invalidating iterators.

[!] Programmers need to be aware of this and must either re-calculated saved iterator positions or rather used indices.

Iterators vs. Indices

Technically indices might have a slight performance penalty compared to pointers and iterators, as an addition (base address plus offset) is required at run-time.

> On modern hardware performance effects are negligible or invisible,* as these additions are usually carried out in special hardware units, while decoding the relevant machine operations.

Using indices instead of iterators to permanently denote the position of *some container elements of special interest* should be strongly considered

- for `std::vector` (and `std::basic_string` including its type aliases)
- if the content changes frequently and may trigger growth beyond the size the container had when the positions were determined.

*: Though there may be indirect effects as register usage tends to grow, hence reducing the number of registers available for temporaries.

Iterators vs. Range-For

```
container data;
for (auto e : data) // 1
    …
for (auto &e : data) // 2
    …
for (const auto &e : data) // 3
    …
```

Iterators are neither superior nor inferior to range-`for` loops.

The code shown left (with `container` replaced by any STL container instantiated for a concrete type) …*

… is equivalent to the code shown below:

```
for (auto it = std::begin(data); it != std::end(data); ++i) { // 1
    auto e = *it; …
}
for (auto it = std::begin(data); it != std::end(data); ++i) { // 2
    const &e = *it; …
}
for (auto it = std::begin(data); it != std::end(data); ++i) { // 3
    const auto &e = *it; …
}
```

*: Actually this is true for any (also non-STL) container and the global overloads are automatically mapped to member functions of the same name (but without arguments).

"Big-O" Performance Estimation

Please follow the verbal explanations given by trainer, based on the adjoining Info-Graphic ...

... and feel free to ask any question you want to ask.