C++11 BLWS (Wednesday 2)

The STL and Beyond

Short breaks will be inserted as convenient.

See also Josuttis: 6.2

STL Containers (recap)

The STL and its template-based containers are at the heart of reusable library components standardized with C++98 (besides character strings and I/O-streams).

While there originally where

- three sequential containers, and
- four associative containers,

C++11 had extended this by

- two more to a **total of five sequential containers**, and
- four more to a **total of eight associative containers**.

Sequential Containers

Sequential containers more or less reflect typical basic data structures:

- Contiguous memory …
  - … of fixed size: `std::array` (new in C++11)
  - … with heap allocation: `std::vector`
- Lists were elements point to each other …
  - … singly linked: `std::forward_list` (new in C++11)
  - … doubly linked: `std::list`
- Finally a mix of both in form of …
  - … adjacent memory in chunks: `std::deque`*

*: Abbreviating *Double Ended Queue* but pronounced as "deck".

Choosing from Sequential Containers

The choice largely depends on the required operations and their performance.

- When **size can be fixed at compile time**, `std::array` is best.*
- If **insertions and extractions need to occur at both or at different ends**, this leaves `std::list` and `std::deque` …
- … otherwise `std::vector` would be **a good default choice**.
- Furthermore `std::array` and `std::vector` are the only to choose from when **memory layout compatible to native arrays** is mandatory.
- If **random access is required** `std::array`, `std::vector` and `std::deque` are the candidates …
- … otherwise, if **insertions and extractions need to be efficient everywhere** `std::list` and `std::forward_list`.

---

*: The picture is slightly different if only the *maximum size* can be fixed at compile time: depending on what is to be stored, an `std::array` may rule out itself when there is no way to tell "valid" from "invalid" elements and there is no suitable (default) constructor for the latter ones.

See also Josuttis: 7.7-7.8

Associative Containers

Associative containers can be grouped around three dimensions:

- Whether they contain **a key value pair or just a key**:
  - `map`-types versus `set`-types
- Whether **duplicate keys are allowed**:
  - `multi`-types versus the others.
- Whether the **underlying data structures is hash-based or a binary tree**:
  - `unordered`-types versus the others.

Viewed pragmatically the data structure rather impacts the following:

- Need lookups be as fast as possible (O(1) even for huge collections)?
- Will there be sequential processing with preferred order of traversal?

If the first is more important as the second, prefer hash-based variants, if vice versa the tree-based ones.

See also

STL Container Adapters

There are a number of use patterns for which the standard STL containers are more powerful as desirable – or in other words:

- where some operations should be completely taken away,
- or be made available in a limited or different form.

This is the problem that adapters are designed to solve.

Stacks

The `std::stack` adapts a container so that insertions and deletions can take place only at the same end.

- The default underlying container is an `std::deque` but also `std::vector` or `std::list` can be chosen.*
- Instead of "back" and "front" as conventionally used for a sequential container
  - the most recent element (inserted last) is named "top", and
  - at that end elements are simply "push"-ed and "pop"-ed.
- Also there is no iterator style (or other) interface to iterate over the whole content.

---

*: Interestingly an `std::forward_list` cannot be adapted to a stack, though it implements perfect stack semantics. This is because a stack adapter expects to work at the "back"-side of its underlying container, not the "front"-side that the singly linked list exposes.

Stack Code Examples

The following shows a possible way to fill …

```
std::stack<int> data;
…
int x;
while (std::cin >> x)
    data.push(x);
```

… and to extract from a stack;

```
while (!data.empty()) {
    std::cout << data.top() << ' ';
    data.pop();
}
```

See also Josuttis: 12.2

Queues

The `std::queue` adapts a container so that insertions and deletions can take place only at different ends.

- The default underlying container is an `std::deque` but also `std::list` can be chosen.
- The "back" and "front" ends as conventionally used for a sequential container are kept, with
  - the most recent element (inserted last) is "push"-ed to the "back", and
  - the least recent element (inserted longest ago) may be "pop"-ed from the "front".
- Also there is no iterator style (or other) interface to iterate over the whole content.

Queue Code Examples

The following shows a possible way to fill …

```
std::queue<int> data;
…
int x;
while (std::cin >> x)
    data.push(x);
```

… and to extract from a queue;

```
while (!data.empty()) {
    std::cout << data.front() << ' ';
    data.pop();
}
```

Priority Queues

Considering its core interface an `std::priority_queue` is similar to an `std::queue`.

- The default underlying container is an `std::vector`.
- Elements are "push"-ed and "pop"-ed …
  - … while any extraction takes place on the "top" …
  - … which is the largest element currently contained.
- For flexibility the sort criteria may be specified at construction time.

> Priority queues can be – and probably are – implemented based on the STL algorithms for managing heap data structures.*

*: Here the term heap is used differently from its usual meaning, which is to refer to the *free store area* of the (technical) execution model.

Priority Queue Code Examples

A priority queue storing `int`-s in the default order (extract greatest first) …

```
std::priority_queue<int> pq1;
```

… which is the same as specifying all defaults explicit …

```
std::priority_queue<int, std::vector<int>, std::less<int>> pq1;
```

… and vice versa (extracts smallest first):*

```
std::priority_queue<int>, std::vector<int>, std::greater<int>> pq2;
```

Note that the comparison functor orders by what goes to the top **last**!

*: As the sort functor is the third template argument, the second (the container on which the priority queue is based) must be specified too, even if the default applies.

C++11: New Containers

The following were introduced with C++11:

- `std::array` – a wrapper around native arrays
- `std::forward_list` – a singly linked list
- `std::unordered_set` – the hash-based version of `std::set`
- `std::unordered_multiset` – the hash-based version of `std::multiset`
- `std::unordered_map` – the hash-based version of `std::map`
- `std::unordered_multimap` – the hash-based version of `std::multimap`
- `std::bitset` is not a new container* but has been slightly extended.

The iterator interface of all existing containers has been extended with the member functions `cbegin()`, `cend()`, `crbegin()`, and `crend()` to obtain `const_iterator`s from non-`const` containers.

---

*: From a language lawyer point of view an `std::bitset bs` may not even be called "container" as it misses the standard iterator interface – only counting loops (`for (i = 0; i < bs.size(); ++i) …`) are supported.

std::array

The `std::array` wrapper makes native arrays look more like standard conforming containers. It has to be instantiated with a type and a size argument:*

```
std::array<double, 50> data;
```

The wrapper adds:

- Standard embedded type definitions like `size_type`, `value_type` etc.
- Standard member functions like `size()`, `empty()` etc.
- Everything required for walking over the content using the standard iterator interface.

The latter not only makes iterating over native arrays no more different from the other containers, but also enables range based for loops.

On the other hand, native arrays are compatible with range based loops, so – if this were the only reason to use the wrapper – it actually isn't necessary!

*: The size argument cannot be omitted, even if the array definition is combined with an initialization!

std::forward_list

This container makes a minimum-overhead, singly-linked list available.

It has the following differences to the probably well-known `std::list`:

- An `std::forward_list` does not know its size – so testing for contained elements must be with `empty()`, not `size() == 0`.
- Insertion and extraction take place at the "front"-side, hence
  - member functions `front`, `push_front`, and `pop_front` are available
  - but not their counterparts working at the "back"-side.
- When walking over the content with an iterator, it does not allow to insert new elements at the iterator position but only after it.
  - This is technically easy to explain as a minimum-overhead iterator can not know the element prior to the one it refers to.
  - Therefore the member function implementing insertion is not called `insert` but `insert_after`.

See also Josuttis: 7.9

Hash-based Containers

Hash-based containers will improve lookup-performance at the price that the order, when iterating over the container, is not predictable:

- With a decent hash function most any key will be found in O(1) time – or a decision made that a key does not exist.
- When the container is filled by incrementally adding more elements over time, it also will extend its reserved space incrementally and therefore its elements must more or less frequently be re-hashed.
- Differently from `std::vector` iterators keep valid even when elements are rehashed.
- Though no specific order can be expected when walking over a hash-based container, elements with the same key follow each other (as elements with the same hash-value will do).
- Elements of a set or keys of a map are not required to implement `operator<` but `operator==`.*

---

*: Though probably not to expect in practice this could have subtle effects if an ordered container were exchanged with its unordered cousin or vice versa.

Boost: Unordered

If a C++ implementation does not (yet) make hash-based containers available Boost.Unordered may be used as drop-in replacement.

> The most important difference is that the standard containers are in namespace `std` while their Boost couterparts are in namespace `boost`.

Boost: BiMap

Boost.Bimap allows bidirectional key-value lookup.

```
class IP { … };   // classes for objects to map bidirectionally
class Host { … }; // to each other (also works with built-in types)
…
typedef boost::bimap<IP, Host> IP_to_Host; // define type ...
IP_to_Host nameservice;                    // ... and object
…
// fill nameservice (bidirectional map) with content:
nameservice.insert(IP_to_Host::value_type(IP(…), Host(…)));
nameservice.insert(IP_to_Host::value_type(IP(…), Host(…)));
…
// lookup now works both ways:
… nameservice.left.find(IP(…)) …    // lookup entry via IP-Number
… nameservice.right.find(Host(…)) … // lookup entry via Host-Name
```

The interface to

- `nameservice.left` is like that of a `map<IP, Host>`;
- `nameservice.right` is like that of a `map<Host, IP>`.

Boost: Property Tree

Boost.Property_Tree is a recursive data structure.

A node (starting with the single root node)

- has some data value (typically an `std::string`)
- and a list of children (empty for leaf nodes),
  - each of which holds a key (typically an `std::string`)
  - and a sub-tree starting with a node that
    - has some data value
    - and a list of children
      - each of which holds a key
      - and a sub-tree starting with …

Property Tree (cont.)

When represented in main memory a property tree can be navigated programmatically (e.g. searched for sub-notes).*

Besides that in can be made persistent in an external representation, i.e. written to a file and read back in a number of serialization formats:

- JSON (JavaScript Object Notation)
- Info-File format (similar to JSON with lesser punctuation)
- Ini-File format (as widely used by MS-Windows "in the old days")
- XML (with a fixed conversion schema, no generic DTD or document object model)

Therefore property trees are well suited for storing configuration data or state information between consecutive program executions, though that's not their sole possible use.

---

*: Navigation is only within one level, there is no automatic recursion, i.e. any descending into a sub-tree must be explicit, it will not happen automatically.

C++11 Extensions to Bitsets

The `std::bitset` class from C++98 has been extended in C++11 with respect to

- conversion to the (now officially supported) type `long long`,
- a hash-function so that instances may be used in the unordered-variants of associative containers,
- optional arguments for the `to_string` conversion, and
- a constructor accepting a `std::string` argument.

One effect that can be achieved easily with the latter two extensions is to convert numbers to and from a binary representation as text form, with freely chosen characters to denote set and cleared bits.

Binary Literals

With the macro `BOOST_BINARY` (defined in `<boost/utility.hpp>`) it becomes possibilitiy to specify binary literals as series of `0` and `1`:*

- `BOOST_BINARY(1 1 1 1)` yields `0xF` or `15`
- `BOOST_BINARY(1 0 1 0 1 0 )` yields `0x1A` or `42`
- `BOOST_BINARY_UL(1 0 1 0 1 0 )` yields `0x1AL` or `42uL`
- `BOOST_BINARY(100 0000 0000)` yields `0x400` or `1024`

---

*: In C++14 internal support for binary literals is expected so that examples like above could be written as `0b1111`, `0b101010` etc. Some C++ compilers already support this as an extension.

STL Iterators (recap)

The STL was designed around the idea that the glue between C++ Containers and C++ Algorithms should be provided by Iterators which were further categorized by their capabilities, leading to C++ Iterator Categories:

- **Input- and Output-Iterators** – same operations as Forward-Iterators (see next) but with restrictions on the order of operations.
- **Forward-Iterators** – comparable to each other, `operator++` in pre- and postfix version, `operator*` for dereferencing.
- **Bidirectional-Iterators** – in addition `operator--` in pre- and postfix version.
- **Random-Access-Iterators** – in addition `operator+=` and `operator-=` to increment and decrement in steps of more than one and `operator-` to determine the distance between two iterators.

Containers document the iterator category they **provide** and **Algorithms** the category they **require**.

STL Iterators Efficiency

To allow a maximally efficient implementation of iterators with respect to

- Memory Footprint and
- Runtime Performance

safety requirements were not cast into any of the C++ standards but left as a *Quality of Implementation* (QoI) issue.[*]

For most containers an iterator can be as efficient as a pointer.

Walking through all elements of a container with an iterator does

- not **require** any large support data structure, and
- **may** map to very few machine instructions at assembler level.

---

[*]: What is frequently misunderstood is that implementations actually **can** prefer safety over efficiency; some versions of the STL have special macros that can be defined to enable additional tests, e.g. _STLP_DEBUG for STLport. If other vendors (or free implementations) of Standard C++ do not exploit the leeway given by the standard to add "safety" features, then presumably because there is not that much pressure from the respective clients to have such features ... or at least not at the price to pay for it in terms of lost efficiency.

STL algorithms (recap)

Algorithms originally supplied when the STL was standardized with C++98 are too numerous to give them full coverage here.

Instead refer to a good documentation source like named below or equivalent:

- http://en.cppreference.com/w/cpp/algorithm
- http://www.cplusplus.com/reference/algorithm/

The following only recapitulates some of the systematic approach and after this highlights a few C++11 extensions.

Iterator Interface to Containers

Algorithms usually deal with containers but abstract that fact away by taking two iterator arguments:

```
std::vector<int> data;
…
// add all values in container
const auto n = std::accumulate(data.begin(), data.end(), 0);
```

This may seem as a minor inconvenience but easily allows for subranges without the need for specialized algorithms:

```
assert(data.size() > 3);
// add values in container except the first two and the last
const auto n = std::accumulate(data.begin()+2, data.end()-1, 0);
```

Note that a typical requirement of algorithms is that: of the two iterators specifying the range the second must be *reachable* from the first.

This means, after the first is incremented often enough it must – while staying within the container bounds – compare equal to the second.

Generic Iterator Interface

Another reason that makes the iterator interface to algorithms powerful is that iterators are not required to connect to containers.

> It can make sense to define a class that exposes the same interface as an iterator while actually calculating values on the run.

```
class Iota
    : public std::iterator<std::input_iterator_tag, int> {
    int i, s;
public:
    Iota(int i_ = 0, int s_ = 1) : i(i_), s(s_) {}
    int operator*() { return i; }
    Iota operator++() { return Iota(i += s, s); }
    Iota operator++(int) { i += s; return Iota(i - s, s); }
};
```

Given the above class a container can be filled with a number range:

```
std::copy_n(Iota(), 20, std::back_inserter(data));
std::copy_n(Iota(10, 3), 20, std::back_inserter(data));
```

Algorithm Families

To the degree to which it makes sense STL algorithms come in families:

- A *plain algorithm* (say: `std::remove`) does its work modifying a container, basing its decision (which elements to remove) on a fixed value.

- The `_copy`-version (i.e. `std::remove_copy`) leaves the original container unmodified, storing the result to another container*

- The `_if`-version (i.e. `std::remove_if`) generalizes the predicate through using a *Callable* as last argument (function, functor, lambda).

- If both of the above versions exist they will also be found combined into a `_copy_if` version (i.e. `std::remove_copy_if`).

---

*: The target for this operation again is specified with an *iterator* that gets used as Output Iterator; usually it will be kind of an inserter, i.e. `std::back_insert_iterator` or `std::front_insert_iterator` for sequential containers, `std::insert_iterator` for associative containers, or `std::ostream_iterator` for output streams.

Using Algorithms on Native Array

With native pointers used as iterators, all of the STL algorithms can also be used to process native arrays.

> There is price to pay: any algorithm that logically removes values cannot make the array smaller – so it returns the new end.

```
int data[1000], *data_end = data;
… // fill with values, make data_end point behind the last
std::sort(data, data_end);               // first sort then ...
data_end = std::unique(data, data_end);  // ... remove duplicates
```

Alternatively the `std::array` wrapper could be applied:

```
std::array<int, 1000> data;
auto data_end = data.begin();
… // fill and make data_end to refer after the last ...
std::sort(data.begin(), data_end));
data_end = std::unique(data.begin(), data_end);
```

Return Values from Removals

Generally algorithms that logically remove elements from a container return the "new end".

This is also true if the physical size of the underlying container could be made smaller, like for the sequential STL containers:

```
// assuming a container x with elements of type int:
auto new_end = std::remove_if(x.begin(), x.end(),
                              [](int e) { return (e < 42); });
```

Physically reducing the container size must happen explicitly:

```
x.erase(new_end, x.end());
```


Return Values from Filling

Many algorithms that fill a container – in case of families particularly the `_copy`-version – return an iterator pointing after the last element copied to the result.

This is often useful, especially when the filling state of a native array needs to be tracked:*

```
const int N = 32767;
static_assert(ckdiv2(N+1), "N must be one less than a power of two");
int data[N], *data_end = data;
for (int n = 1, i = (N+1)/2; i > 0; ++n, i /= 2)
    data_end = std::fill_n(data_end, i, n);
```

---

*: To figure out how exactly that code initializes the container with which values is left as an exercise for the reader :-) … if not from a code analysis then go and wrap the above fragment into a `main`-function, compile, run, and look at the output … (maybe reduce `N` to much smaller values, say 7, 15, 31, …).

C++11: New Algorithms

Again the following does not intend to be exhaustive, instead it should just shortly highlight C++11 additions for those who know STL algorithms quite well from C++98.

- The largest group is the one in which most algorithms start with `is_`. These check for certain properties of a container content.[*]
- A few algorithms have probably been added because a substantial number of people found them "missing".
  - In one subgroup the new algorithms will actually improve performance.
  - Others could have been well expressed with the available algorithms but an algorithm with an expressive name makes the intent clearer.

---

[*]: Or rather sequences, as the interface is of course through iterators, so no underlying container needs to be exposed.

New Ways to Search through Containers

The following algorithms take a predicate as argument to check for **each** container element, but all have the chance to shortcut the evaluation:

- `std::all_of` – can return `false` the first time the predicate is `false`
- `std::any_of` – can return `true` the first time the predicate is `true`
- `std::none_of` – can return `false` the first time the predicate is `true`

> Though any of the above can be written in terms of any other, it is usually clearest to express the intent with the fewest negations.

Compare:

```
std::none_of(x.begin(), x.end(), [](int v) { return (v > 0); }
!std::any_of(x.begin(), x.end(), [](int v) { return (v > 0); }
std::all_of(x.begin(), x.end(), [](int v) { return !(v > 0); }
```

---

*: For a good test whether a condition is clearly expressed imagine you read it out to someone at the telephone. Which one of these do you think will be understood then without any doubt or asking back?

New Ways to Fill Containers

A new algorithm to copy from one container to another one allows to specify the number of elements instead of an end iterator.

This allows sometime for easier calculations:

```
const std::size_t Ndata = 50, Nresult = 20;
int data[Ndata], *data_filled = data;
int result[Nresult], *result_filled = result;
… // copy to data, assume it is finally filled up to data_filled
… // also assume result has now content up to result_filled
…
// copy from data to result until data exhausted or result full:
const auto available = data_filled - data;
const auto freespace = result + Nresult - result_filled;
std::copy_n(data, std::min(available, freespace), result_filled);
```

Also it is now easily possible to fill a container with increasing values:*

```
std::iota(data, data+Ndata, 1); // fills in: 1 2 3 4 ...
```

*: Note that `std::iota` **expects container elements to exist**. E.g. if an `std::vector` had to be filled it must be resized first.

Algorithms Adding Negation

Elements to be copied from one container to another one can now be selected with a predicate.

Given two sequential containers `x` and `result` with element type `T` the following code would do this:*

```
std::copy_if(x.begin(), x.end(), std::back_inserter(result),
             [](T elem) {
                 return …; // whatever needs to be checked for elem
             });
```

With a negated predicate `std::remove_copy_if` had the same effect.

Similarly when searching for the first container element that does not have a given property, checked with a predicate:

```
… std::find_if_not(x.begin(), x.end(), [](T elem) { return …; });
```

Again, `std::find_if` with a negated predicate did the same search.

---

*: As `T` is handed to the lambda by value, it must be copyable; alternatively the argument can be handed over as (`const`) reference.

Algorithms Improving Performance

It is now possible to search for the minimum and maximum value at the same time, passing over the sequence only once with fewer comparisons:

```
auto result_it = std::minmax_sequence(x.begin(), x.end());
… result_it.first … // iterator denoting minimum element
… result_it.last … // iterator denoting maximum element
```

> **[!]** Of course, the iterators returned must not be dereferenced without a prior test if the sequence might have been empty!

There is also a simpler version returning a reference to two variables:

```
int a, b;
… // fill in values
auto result = std::min_max(a, b);
… result.first … // reference to variable holding the smaller value
… result.last … // reference to variable holding the larger value
// or store into two separate variables:
int min, max;
std::tie(min, max) = std::minmax(a, b);
```

Boost: Range

Boost.Range extends the basic idea to combine iterators to pairs – called ranges – with the goal to eliminate the need to always have to supply two arguments when just a single container is involved.*

- Additional algorithms accept these ranges instead of the separate arguments their STL counterparts require.
- The concept of ranges is further extended so that algorithms can be glued together with a high degree of flexibility.

---

*: Because the two arguments style adopted by the original STL has the advantage that subranges do not require additional algorithms, ranges from this library can of course not only be constructed from containers (taking `begin()` and `end()` as range) but also from two container iterators!

Boost: Algorithm

Boost.Algorithm is a collection of algorithms to extending the STL.

Parts of this collection have been standardized with C++11, others are intended to become part of C++14, some will still stay "Boost only".