Back to Basics: Classic STL

Bob Steagall
CppCon 2021
Overview

• Rationale

• History and design overview

• Iterators

• Containers

• Algorithms
Goals and References

• Goals
  • Understand overall STL design
  • Understand iterators

• Recommended references
  • *The Standard C++ Library, Second Edition*
    Nicolai M. Josuttis – Addison-Wesley 2012

  • *Effective STL*
    Scott Meyers – O'Reilly 2001

  • *Programming: Principles and Practice Using C++, Second Edition*
    Bjarne Stroustrup – Addison-Wesley 2014

  • cppreference.com
What is "Classic STL?"

The C++20 Standard Library

- Language Support
- Concepts
- Diagnostics
- Strings
- Ranges
- General Utilities
- Containers
- Iterators
- Algorithms
- Input/Output
- Atomic Operations
- Thread Support
- Numerics
- Time
- Localization
- Regular Expressions
What is "Classic STL?"

• The short answer – containers + iterators + algorithms + some utilities
Rationale
Rationale

• We have some business problem to solve
• We begin with input data
• We read that data and perform computations
• We generate and write some desired output
Rationale

- Data is almost always *collections* of *elements*
  - A virtually infinite number of data element types

- Each collection of elements has some *representation*
  - A large number of possible representations

- There are many kinds of processing (*algorithms*)
  - A very large number of algorithms

- In any given problem space, the choices are fewer
  - Call them $N_T$, $N_R$, and $N_A$
  - Traditionally, a combinatorial explosion of code – $N_T \times N_R \times N_A$

- We'd like a smaller number – $N_T + N_R + N_A$ – *this is the goal of the STL*
History and Overview of the STL
A Brief STL History

- 1979, Alexander Stepanov begins exploring generic programming (GP)
- 1988, Stepanov and David Musser publish *Generic Programming*
Generic programming centers around the idea of **abstracting from concrete**, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software.

— David Musser, Alexander Stepanov

*Generic Programming* (1988)

[emphasis mine]
Generic Programming

Following Stepanov, we can define generic programming without mentioning language features: **Lift algorithms and data structures from concrete examples to their most general and abstract form.**

— Bjarne Stroustrup


[emphasis mine]
A Brief STL History

• 1979, Alexander Stepanov begins exploring generic programming (GP)

• 1988, Stepanov and David Musser publish *Generic Programming*

• 1992, Meng Lee joins Stepanov at HP Research Labs, where his team is experimenting with C and C++

• 1993, Stepanov presents the main ideas at the November WG21 meeting

• 1994, Stepanov and Lee create proposal for WG21 that was accepted later that year

• 1994-1998, much additional work; adding the original associative containers

• 1998, first ISO C++ Standard published

• 2011, C++11 is published, and with some new containers
Original Design Principles

- Comprehensive
  - Take all the best from APL, Lisp, Dylan, C library, USL Standard Components...
  - Provide structure and fill the gaps

- Extensible
  - Orthogonality of the component space
  - Semantically based interoperability guarantees

- Efficient
  - No penalty for generality
  - Complexity guarantees at the interface level

- Natural
  - C/C++ machine model and programming paradigm
  - Support for built-in data types
Original Design Principles

- Comprehensive
  - Take all the best from APL, Lisp, Dylan, C library, USL Standard Components...
  - Provide structure and fill the gaps

- Extensible
  - Orthogonality of the component space
  - Semantically based interoperability guarantees

- Efficient
  - No penalty for generality
  - Complexity guarantees at the interface level

- Natural
  - C/C++ machine model and programming paradigm
  - Support for built-in data types
Complexity and the Big-O Notation

- **Complexity** refers to the *runtime cost* of an algorithm

- Big-O notation expresses the *relative complexity* of an algorithm

<table>
<thead>
<tr>
<th>Type</th>
<th>Notation</th>
<th>Runtime Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>O(1)</td>
<td>Independent of number of elements</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>O(log(n))</td>
<td>Increases logarithmically with the number of elements</td>
</tr>
<tr>
<td>Linear</td>
<td>O(n)</td>
<td>Increases linearly with the number of elements</td>
</tr>
<tr>
<td>N-log-N</td>
<td>O(n*log(n))</td>
<td>Increases as a product of linear and logarithmic complexities</td>
</tr>
<tr>
<td>Quadratic</td>
<td>O(n²)</td>
<td>Increases as the square of the number of elements</td>
</tr>
</tbody>
</table>
Key Principles

- **Containers** store *collections* of *elements*

- **Algorithms** perform operations upon collections of elements

- Containers and algorithms are entirely independent

- **Iterators** provide a common unit of information exchange between containers and algorithms

```
Container
{e_0, e_1, e_2, ...}  

Iterator

Algorithm
```
Key Principles

- **Containers** store *collections* of *elements*

- **Algorithms** perform operations upon collections of elements

- Containers and algorithms are entirely independent

- **Iterators** provide a common unit of information exchange between containers and algorithms
Complexity and Interfaces

- STL makes complexity guarantees by specifying *interfaces* and *requirements*

- Containers provide support for
  - Adding / removing elements
  - Accessing (reading / updating) elements via associated iterators
  - A container's iterators understand (and *abstract*) that container's internal structure

- Iterators
  - Provide access to container elements through well-defined interfaces with strict guarantees

- Algorithms
  - Employ the well-defined interfaces provided by iterators
  - Have complexity based on the algorithm itself and the guarantees made by the iterators
Containers Overview

- Containers hold a collection of elements
  - STL containers are implemented using a variety of basic data structures
  - Each STL container represents a **sequence** of elements

- Containers have an internal structure and ordering
  - We can observe this ordering
  - Sometimes we can control the ordering

- **Containers own the elements they hold**
  - Ownership means element lifetime management
  - Containers construct and destroy their member elements
Containers Overview

- Sequence containers
  - vector
  - deque
  - list
  - array (C++11)
  - forward_list (C++11)

- Associative containers
  - map
  - set
  - multimap
  - multiset

- Unordered associative containers
  - unordered_map (C++11)
  - unordered_set (C++11)
  - unordered_multimap (C++11)
  - unordered_multiset (C++11)

- Container adaptors
  - queue
  - stack
  - priority_queue
Iterators Overview

- Iterators typically provide a way of observing a container's elements and ordering
  - Some containers provide more than one way to observe elements

- Iterators *may* provide a way of modifying a container's elements

- An iterator's interface specifies
  - The complexity of observing and traversing a collection's elements
  - The manner in which elements are observed
  - Whether an element can be read from or written to

- **Iterators never own the elements to which they refer**
Iterators Overview

- Classic STL has five iterator categories
  - Output
  - Input
  - Forward
  - Bidirectional
  - Random-access

- Arranged in a hierarchy of *requirements*
  - **Not** public inheritance
Algorithms Overview

• The algorithms process ranges of elements of a collection
  • Require at least one explicitly-specified iterator pair

• Algorithm categories
  • Non-modifying algorithms
  • Modifying algorithms
  • Removing algorithms
  • Mutating algorithms
  • Sorting algorithms
  • Sorted range algorithms
  • Numeric algorithms
Iterators
Regarding Iterators

• Where do the five iterator categories come from?
• What interface does each category provide?
• What is their time complexity?
• How are they related to containers?
• How are they used by the algorithms?
• Let's try a generic programming exercise and develop iterators from scratch
Referring to Elements in Arrays

- Consider pointers to 2 elements in an array of N objects
  - What can you do with them?

<table>
<thead>
<tr>
<th>Action</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access element</td>
<td>*p</td>
</tr>
<tr>
<td>Access member of element</td>
<td>p-&gt;mem</td>
</tr>
<tr>
<td>Compare for equality of position</td>
<td>p == q, p != q</td>
</tr>
<tr>
<td>Move forward by 1</td>
<td>++p, p++</td>
</tr>
<tr>
<td>Move backward by 1</td>
<td>--p, p--</td>
</tr>
<tr>
<td>Make a copy (assign)</td>
<td>q = p</td>
</tr>
<tr>
<td>Access arbitrary element</td>
<td>p[n]</td>
</tr>
<tr>
<td>Move forward by arbitrary n</td>
<td>p += n, q = p + n</td>
</tr>
<tr>
<td>Move backward by arbitrary n</td>
<td>p -= n, q = p - n</td>
</tr>
<tr>
<td>Compare for relative position</td>
<td>p &lt; q, p &lt;= q, p &gt; q, p &gt;= q</td>
</tr>
<tr>
<td>Find distance between two elements</td>
<td>d = q - p</td>
</tr>
</tbody>
</table>

\[ \begin{array}{cccccccc}
&e_0& e_1& \ldots & e_{p-1} & e_p & e_{p+1} & \ldots & e_{q-1} & e_q & e_{q+1} & \ldots & e_{N-1} \\
\end{array} \]

\[ p \quad q \]

\[ O(1) - \text{constant time!} \]
Referring to Elements in Doubly-Linked Lists

- Consider pointers to 2 nodes in a simple doubly-linked list

<table>
<thead>
<tr>
<th>Action</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access element</td>
<td>*p</td>
</tr>
<tr>
<td>Access member of element</td>
<td>p-&gt;mem</td>
</tr>
<tr>
<td>Compare for equality of position</td>
<td>p == q, p != q</td>
</tr>
<tr>
<td>Move forward by 1</td>
<td>p = p-&gt;next</td>
</tr>
<tr>
<td>Move backward by 1</td>
<td>p = p-&gt;prev</td>
</tr>
<tr>
<td>Make a copy (assign)</td>
<td>q = p</td>
</tr>
</tbody>
</table>

• What can you do with them?

![Diagram of doubly-linked list with pointers to nodes](image)

O(1) - constant time
Referring to Elements in Singly-Linked Lists

- Consider pointers to 2 nodes in a simple singly-linked list and

\[
\begin{array}{c}
\text{lst} \\
\text{next} \\
\text{ef} \\
\text{next} \\
\text{...} \\
\text{next} \\
\text{ep} \\
\text{next} \\
\text{...} \\
\text{next} \\
\text{eq} \\
\text{next} \\
\text{...} \\
\text{null} \\
\text{el} \\
\text{next} \\
\text{...} \\
\text{next} \\
\text{p} \\
\text{next} \\
\text{q} \\
\text{...} \\
\text{next} \\
\text{...} \\
\text{next} \\
\text{null} \\
\end{array}
\]

- What can you do with them?

<table>
<thead>
<tr>
<th>Action</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access element</td>
<td>*p</td>
</tr>
<tr>
<td>Access member of element</td>
<td>p-&gt;mem</td>
</tr>
<tr>
<td>Compare for equality of position</td>
<td>p == q, p != q</td>
</tr>
<tr>
<td>Move forward by 1</td>
<td>p = p-&gt;next</td>
</tr>
<tr>
<td>Make a copy (assign)</td>
<td>q = p</td>
</tr>
</tbody>
</table>

$O(1)$ - constant time
Multi-Pass and Single-Pass Iteration

• Arrays, doubly-linked and singly-linked lists all support *multi-pass iteration*
  • Pointers to elements can be dereferenced more than once, with the same result each time
  • The sequence can be iterated over (traversed) more than once

• What about sequences that can be traversed only once?
  • Some sequences support only *single-pass iteration*
  • An element can only be read from, or written to, a given position one time
  • The act of reading or writing irrevocably changes position
  • Reading from / writing to file streams, sockets, raw devices, etc.
Reading Elements (Bytes) From a FILE Stream

- Consider a pointer to a FILE stream opened for input

```c
FILE* p = fopen(name, "rb")
```

- What can you do with it?

<table>
<thead>
<tr>
<th>Action</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read element and advance</td>
<td>b = fgetc(p)</td>
</tr>
<tr>
<td>Compare for end-of-file equality</td>
<td>b == EOF, feof(p)</td>
</tr>
<tr>
<td>Make a copy (assign)</td>
<td>q = p</td>
</tr>
</tbody>
</table>

- $O(1)$ - constant time
Writing Elements (Bytes) To a FILE Stream

• Consider a pointer to a FILE stream opened for output

```
FILE* p = fopen(name, "wb")
```

- What can you do with it?

<table>
<thead>
<tr>
<th>Action</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write element and advance</td>
<td>fputc(b, p)</td>
</tr>
<tr>
<td>Make a copy (assign)</td>
<td>q = p</td>
</tr>
</tbody>
</table>

\[ O(1) - \text{constant time} \]
Iterator Categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Write forward, single-pass</td>
</tr>
<tr>
<td>Input</td>
<td>Read forward, single-pass</td>
</tr>
<tr>
<td>Forward</td>
<td>Access forward, multi-pass</td>
</tr>
<tr>
<td>Bidirectional</td>
<td>Access forward and backward, multi-pass</td>
</tr>
<tr>
<td>Random Access</td>
<td>Access arbitrary position, multi-pass</td>
</tr>
</tbody>
</table>

- Arranged in a hierarchy of requirements
  - Not public inheritance
  - Arrow to X means: "satisfies at least the requirements of X"
  - Dotted arrow means: "optional"

- Iterators that satisfy the requirements of output iterators are called *mutable* iterators
• Let's think about sequences in terms of *positions*
  
  • By fiat, a sequence of N elements has N+1 positions
  • The first N positions contain elements and are *dereferenceable*
  • Assume the last position contains nothing and is therefore *non-dereferenceable*
  • You can point/refer to the last position, but you cannot read from it or write to it
Iterator Ranges

- In the STL, iteration over sequences is based on the idea of *iterator ranges*

- An iterator range is represented by a pair of iterators -- \([\text{first}, \text{last})\)
  - This pair represents a *half-open interval* over the sequence of elements
  - \text{first} refers to the first element *included* in the sequence
  - \text{last} refers to the non-dereferenceable, "one-past-the-end" (PTE) position *excluded* from the sequence
Iterator Ranges

• Q: Why use ranges described by half-open intervals?

• A: It makes testing for loop termination very simple
  • Loops only need to test for iterator equality
  • Indexing not required
  • Location in memory is irrelevant

```cpp
iterator  f = get_position_of_first_element_in_sequence();
iterator  l = get_one_past_end_position_in_sequence();

// Works for all iterator types except OutputIterator
for (; f != l; ++f) {
    some_function(*f);
}
```
Iterator Ranges

• Q: How can they work?

• A: It depends on the container / sequence
  • Containers that store elements contiguously in memory rely on ability to get a pointer to the "next-position-after"

```cpp
Foo f;
Foo* fb = &f;
Foo* fe = fb + 1;

Foo a[10];
Foo* ab = &a[0];
Foo* ae = ab + 10;
```
Iterator Ranges

Q: How can they work?

A: It depends on the container / sequence
   - Node-based containers can use *sentinel nodes*
Output Iterators – Write Forward, Single-Pass

<table>
<thead>
<tr>
<th>Expression</th>
<th>Action/Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iter q(p)</td>
<td>Copy construction</td>
</tr>
<tr>
<td>q = p</td>
<td>Copy assignment</td>
</tr>
<tr>
<td>*p</td>
<td>Write to position one time</td>
</tr>
<tr>
<td>++p</td>
<td>Step forward, return new position</td>
</tr>
<tr>
<td>++p</td>
<td>Step forward, return old position</td>
</tr>
</tbody>
</table>

- The only valid use of the expression \(*p*\) is on the left side of an assignment statement
- Comparison operators are not required – no end of sequence is assumed
  - Output iterators model an "infinite sink"

- `const_iterator` types provided by STL containers cannot be output iterators – `const_iterators` permit only reading
Input Iterators – Read Forward, Single-Pass

- **p == q** does not imply **++p == ++q**

- The comparison operators are provided to check whether an input iterator is equal to the past-the-end iterator

- All iterators that read values must provide at least the capabilities of input iterators; usually, they provide more

<table>
<thead>
<tr>
<th>Expression</th>
<th>Action/Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iter q(p)</td>
<td>Copy construction</td>
</tr>
<tr>
<td>q = p</td>
<td>Copy assignment</td>
</tr>
<tr>
<td>*p</td>
<td>Read access to element one time</td>
</tr>
<tr>
<td>p-&gt;mem</td>
<td>Read access member of element one time</td>
</tr>
<tr>
<td>++p</td>
<td>Move forward by 1, return new position</td>
</tr>
<tr>
<td>p++</td>
<td>Move forward by 1, possibly return old position</td>
</tr>
<tr>
<td>p == q</td>
<td>Return true if two iterators are equal</td>
</tr>
<tr>
<td>p != q</td>
<td>Return true if two iterators are different</td>
</tr>
</tbody>
</table>

**Expression Action/Result**

| Iter q(p)   | Copy construction |
| q = p       | Copy assignment |
| *p          | Read access to element one time |
| p->mem      | Read access member of element one time |
| ++p         | Move forward by 1, return new position |
| p++         | Move forward by 1, possibly return old position |
| p == q      | Return true if two iterators are equal |
| p != q      | Return true if two iterators are different |

**Output**

**Input**

**Forward**

**Bidirectional**

**Random-access**
Forward Iterators – Access Forward, Multi-Pass

<table>
<thead>
<tr>
<th>Expression</th>
<th>Action/Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iter q(p)</td>
<td>Copy construction</td>
</tr>
<tr>
<td>q = p</td>
<td>Copy assignment</td>
</tr>
<tr>
<td>*p</td>
<td>Access element</td>
</tr>
<tr>
<td>p-&gt;mem</td>
<td>Access member of element</td>
</tr>
<tr>
<td>++p</td>
<td>Move forward by 1, return new position</td>
</tr>
<tr>
<td>p++</td>
<td>Move forward by 1, return old position</td>
</tr>
<tr>
<td>p == q</td>
<td>Return true if two iterators refer to the same position</td>
</tr>
<tr>
<td>p != q</td>
<td>Return true if two iterators refer to different positions</td>
</tr>
<tr>
<td>Iter p</td>
<td>Default constructor, create singular value</td>
</tr>
</tbody>
</table>

- Additional capabilities and guarantees
  - p and q refer to the same position IFF p == q
  - p == q implies ++p == ++q
  - Accessing an element (e.g., *p) does not change the iterator's position
## Bidirectional Iterators – Access Forward/Backward, Multi-Pass

<table>
<thead>
<tr>
<th>Expression</th>
<th>Action/Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iter q(p)</td>
<td>Copy construction</td>
</tr>
<tr>
<td>q = p</td>
<td>Copy assignment</td>
</tr>
<tr>
<td>*p</td>
<td>Access element</td>
</tr>
<tr>
<td>p-&gt;mem</td>
<td>Access member of element</td>
</tr>
<tr>
<td>++p</td>
<td>Move forward by 1, return new position</td>
</tr>
<tr>
<td>p++</td>
<td>Move forward by 1, return old position</td>
</tr>
<tr>
<td>p == q</td>
<td>Return true if two iterators refer to the same position</td>
</tr>
<tr>
<td>p != q</td>
<td>Return true if two iterators refer to different positions</td>
</tr>
<tr>
<td>Iter p</td>
<td>Default constructor, create singular value</td>
</tr>
<tr>
<td>--p</td>
<td>Move backward by 1, return new position</td>
</tr>
<tr>
<td>p--</td>
<td>Move backward by 1, return old position</td>
</tr>
</tbody>
</table>

- Additional capabilities and guarantees
  - p == q implies --p == --q
  - --(++p) == p

- Expression Action/Result
  - Iter q(p) Copy construction
  - q = p Copy assignment
  - *p Access element
  - p->mem Access member of element
  - ++p Move forward by 1, return new position
  - p++ Move forward by 1, return old position
  - p == q Return true if two iterators refer to the same position
  - p != q Return true if two iterators refer to different positions
  - Iter p Default constructor, create singular value
  - --p Move backward by 1, return new position
  - p-- Move backward by 1, return old position

### Diagram
- **Output**
- **Input**
- **Forward**
- **Bidirectional**
- **Random-access**
Random-Access Iterators – Arbitrary Access, Multi-Pass

<table>
<thead>
<tr>
<th>Expression</th>
<th>Action/Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Iter q(p)</code></td>
<td>Copy construction</td>
</tr>
<tr>
<td><code>q = p</code></td>
<td>Copy assignment</td>
</tr>
<tr>
<td><code>*p</code></td>
<td>Access element</td>
</tr>
<tr>
<td><code>p-&gt;mem</code></td>
<td>Access member of element</td>
</tr>
<tr>
<td><code>++p</code></td>
<td>Move forward by 1, return new position</td>
</tr>
<tr>
<td><code>p++</code></td>
<td>Move forward by 1, return old position</td>
</tr>
<tr>
<td><code>p == q</code></td>
<td>Return true if two iterators refer to the same position</td>
</tr>
<tr>
<td><code>p != q</code></td>
<td>Return true if two iterators refer to different positions</td>
</tr>
<tr>
<td><code>Iter p</code></td>
<td>Default constructor, create singular value</td>
</tr>
<tr>
<td><code>--p</code></td>
<td>Move backward by 1, return new position</td>
</tr>
<tr>
<td><code>p--</code></td>
<td>Move backward by 1, return old position</td>
</tr>
</tbody>
</table>

- Additional capabilities and guarantees
  - Emulate pointers
  - Provide operators for iterator arithmetic, analogous to pointer arithmetic
  - Provide relational operators to compare position
# Random-Access Iterators – Arbitrary Access, Multi-Pass

<table>
<thead>
<tr>
<th>Expression</th>
<th>Action/Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>p[n]</code></td>
<td>Access element at nth position</td>
</tr>
<tr>
<td><code>p += n</code></td>
<td>Move forward by n elements (backward if n &lt; 0)</td>
</tr>
<tr>
<td><code>p -= n</code></td>
<td>Move backward by n elements (forward if n &lt; 0)</td>
</tr>
<tr>
<td><code>p + n, n + p</code></td>
<td>Return iterator pointing n elements forward (backward if n &lt; 0)</td>
</tr>
<tr>
<td><code>p - n</code></td>
<td>Return iterator pointing n elements backward (forward if n &lt; 0)</td>
</tr>
<tr>
<td><code>p - q</code></td>
<td>Return the distance between positions</td>
</tr>
<tr>
<td><code>p &lt; q</code></td>
<td>True if p is before q in the sequence</td>
</tr>
<tr>
<td><code>p &lt;= q</code></td>
<td>True if p is not after q in the sequence</td>
</tr>
<tr>
<td><code>p &gt; q</code></td>
<td>True if p is after q in the sequence</td>
</tr>
<tr>
<td><code>p &gt;= q</code></td>
<td>True if p is not before q in the sequence</td>
</tr>
</tbody>
</table>

- Additional capabilities and guarantees
  - Emulate pointers
  - Provide operators for iterator arithmetic, analogous to pointer arithmetic
  - Provide relational operators to compare position
Iterator Adaptors

• Reverse iterators
  • template<class Iter> reverse_iterator;
  • Iterates backward from the end of a sequence to the beginning
  • Models a bidirectional iterator when Iter is bidirectional
  • Models a random-access iterator when Iter is random-access

• Insert iterators (inserters)
  • template<class Container> back_insert_iterator;
  • template<class Container> front_insert_iterator;
  • template<class Container> insert_iterator;
  • Models an output iterator that inserts elements at the back / front / interior of a container
Containers
Containers Overview

• Sequence containers
  • Represent ordered collections where an element's position is independent of its value
  • Usually implemented using arrays or linked lists
  • `vector`, `deque`, `list`, `array*`, `forward_list`*

• Associative containers
  • Represent sorted collections where an element's position depends only on its value
  • Usually implemented using binary search trees
  • `map`, `set`, `multimap`, `multiset`

• Unordered associative containers*
  • Represent unsorted collections where an element's position is irrelevant
  • Implemented using hash tables
  • `unordered_map`, `unordered_set`, `unordered_multimap`, `unordered_multiset`
Common Container Interface

• Every STL container provides a common set of nested type aliases

```cpp
template<...>
class container
{
...

    using value_type = ...
    using reference  = ...
    using const_reference = ...

    using iterator     = ...
    using const_iterator  = ...
    using size_type   = ...
    using difference_type = ...
...
```
Common Container Interface

• Every STL container provides a common set of functions

```cpp
template< ... >
class container
{
    ...

    iterator        begin();
    iterator        end();

    const_iterator  begin() const;
    const_iterator  end() const;

    const_iterator  cbegin() const;
    const_iterator  cend() const;
    ...
```
Common Bidirectional Container Interface

- Bidirectional containers provide additional aliases and functions

```cpp
template< ... >
class bidirectional_container
{

... using reverse_iterator = ...
using const_reverse_iterator = ...

reverse_iterator rbegin();
reverse_iterator rend();

const_reverse_iterator rbegin() const;
const_reverse_iterator rend() const;

const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;
...
```
Sequence Container: Vector

template<class T, class Allocator = allocator<T>>
class vector;

• Features
  • Supports amortized constant time insert and erase operations at its end
  • Supports linear time insert and erase operations in its middle
  • Provides const and mutable random-access iterators
  • Provides const and mutable element indexing
  • Supports changing element values
  • Uses contiguous storage for all element types except bool
Sequence Container: Deque

template<class T, class Allocator = allocator<T>>
class deque;

• Features
  • Supports amortized constant time insert and erase operations at both ends
  • Supports linear time insert and erase operations in its middle
  • Provides const and mutable random-access iterators
  • Provides const and mutable element indexing
  • Supports changing element values
Sequence Container: Array

template<class T, size_t N>
class array;

- Features
  - Manages a fixed-sized sequence of objects in an internal C-style array
  - Provides const and mutable random-access iterators
  - Provides const and mutable element indexing
  - Supports changing element values
  - Uses contiguous storage for all element types
Sequence Container: List

template<class T, class Allocator = allocator<T>>
class list;

• Features
  • Supports constant time insert and erase operations anywhere in the sequence
  • Provides const and mutable bidirectional iterators
  • Supports changing element values
  • Provides member functions for splicing, sorting, and merging
  • Usually implemented as a doubly-linked list
Sequence Container: Forward List

template<class T, class Allocator=allocator<T>>
class forward_list;

• Features
  • Supports constant time insert and erase operations anywhere in the sequence
  • Provides const and mutable forward iterators
  • Supports changing element values
  • Provides member functions for splicing
  • Usually implemented as a singly-linked list
Associative Containers: Set

template<class Key,  
    class Compare = less<Key>,  
    class Allocator = allocator<Key>>

class set;

• Features
  • Supports logarithmic time element lookup
  • Elements of type Key are sorted according to Compare
  • Element values are unique
  • Provides const bidirectional iterators
  • Usually implemented as a binary search tree
template<class Key,
    class Compare = less<Key>,
    class Allocator = allocator<Key>>
class multiset;

• Features
  • Supports logarithmic time element lookup
  • Elements of type Key are sorted according to Compare
  • Element values are not unique
  • Provides const bidirectional iterators
  • Usually implemented as a binary search tree
Associative Container: Map

template<class Key, class Val,
    class Compare = less<Key>,
    class Allocator = allocator<pair<const Key, Val>>>
class map;

• Features
  • Supports logarithmic time lookup of a type Val based on a type Key
  • Elements of type pair<const Key, Val> are sorted according to Compare
  • Key values are unique
  • Provides const and mutable bidirectional iterators
    • Mutable iterators permit the Val member of pair<const Key, Val> to be modified
  • Usually implemented as a binary search tree
  • Can be used as an associative array
Associative Container: Multimap

template<class Key, class Val,
    class Compare=less<Key>,
    class Allocator = allocator<pair<const Key, Val>>>
class multimap;

• Features
  • Supports logarithmic time lookup of a type Val based a type Key
  • Elements of type pair<const Key, Val> are sorted according to Compare
  • Key values are not unique
  • Provides const and mutable bidirectional iterators
    • Mutable iterators permit the Val member of pair<const Key, Val> to be modified
  • Usually implemented as a binary search tree
  • Can be used as a dictionary
Unordered Associative Container: Unordered Set

template<class Key,
    class Hash = hash<Key>,
    class Pred = equal_to<Key>,
    class Allocator = allocator<Key>>

class unordered_set;

• Features
  • Supports amortized constant time element lookup
  • Elements of type Key are stored internally in an order determined by Hash
  • Element values are unique
  • Provides const forward iterators
  • Implemented as a hash table – Hash helps determine ordering, Pred tests Key equivalence
Unordered Associative Container: Unordered Multiset

template<class Key,
    class Hash = hash<Key>,
    class Pred = equal_to<Key>,
    class Allocator = allocator<Key>>
class unordered_multiset;

• Features
  • Supports amortized constant time element lookup
  • Elements of type Key are stored internally in an order determined by Hash
  • Element values are not unique
  • Provides const forward iterators
  • Implemented as a hash table – Hash helps determine ordering, Pred tests Key equivalence
Unordered Associative Container: Unordered Map

template<class Key, class Val,
  class Hash = hash<Key>,
  class Pred = equal_to<Key>,
  class Allocator = allocator<pair<const Key, Val>>>

class unordered_map;

• Features
  • Supports amortized constant time lookup of a type Val based on a type Key
  • Elements are of type pair<const Key, Val>
  • Key values are unique
  • Provides const and mutable forward iterators
  • Implemented as a hash table – Hash helps determine ordering, Pred tests Key equivalence
  • Can be used as an associative array
Unordered Associative Container: Unordered Multimap

```
template<class Key, class Val,
    class Hash = hash<Key>,
    class Pred = equal_to<Key>,
    class Allocator = allocator<pair<const Key, Val>>>

class unordered_multimap;
```

- **Features**
  - Supports amortized constant time lookup of a type `Val` based on a type `Key`
  - Elements are of type `pair<const Key, Val>`
  - `Key` values are **not unique**
  - Provides const and mutable `forward` iterators
  - Implemented as a hash table – `Hash` helps determine ordering, `Pred` tests `Key` equivalence
  - Can be used as a dictionary
Container Adaptor: Stack

template<class T, class Container = deque<T>>
class stack;

• Features
  • Wrapper type that implements a classic LIFO stack
  • Amortized constant time `push()` and `pop()` operations
  • Constant time access to next element with `top()`
  • Works with `vector`, `deque`, `list`, and `forward_list`

• Requirements from `Container`
  • Amortized constant time `push_back()` and `pop_back()` member functions
  • Constant time `back()` member function
Container Adaptor: Queue

template<class T, class Container = deque<T>>
class queue;

• Features
  • Wrapper type that implements a classic FIFO queue
  • Amortized constant time `push()` and `pop()` operations
  • Constant time access to next element with `front()` and last element with `back()`
  • Works with `vector`, `deque`, `list`, and `forward_list`

• Requirements from `Container`
  • Amortized constant time `push_back()` and `pop_front()` member functions
  • Constant time `front()` and `back()` member functions
Container Adaptor: Priority Queue

template<class T, class Container = deque<T>>
class priority_queue;

• Features
  • Wrapper type that implements a classic priority queue (AKA heap)
  • Logarithmic time `push()` and `pop()` operations
  • Constant time access to next element with `top()`

• Requirements from `Container`
  • Amortized constant time `push_back()` and `pop_back()` member functions
  • Constant time `front()` member function
  • Random-access iterators (works with `vector` and `deque`)
Algorithms
Algorithms

• There's a large number of algorithms provided by STL (well over 100)
  • Multiple versions of almost all
  • Parallel implementations of some

• Algorithm categories
  • Non-modifying algorithms
  • Modifying algorithms
  • Removing algorithms
  • Mutating algorithms
  • Sorting algorithms
  • Sorted range algorithms
  • Numeric algorithms
Algorithms - Declaration of sort

- sort
  - **Action**: Sorts the elements in the range `[first, last)` in non-descending order; the order of equivalent elements is not guaranteed to be preserved; Elements are compared using the given binary comparison function `comp`
  - **Complexity**: $O(N \cdot \log(N))$, where $N = \text{std::distance}(\text{first}, \text{last})$ comparisons

```cpp
template<class RandomIter, class Compare>
void
sort(RandomIter first, RandomIter last, Compare comp);
```
Algorithms - Declaration of lower_bound

- **lower_bound**
  - **Action**: Returns an iterator pointing to the first element in the range \([\text{first}, \text{last})\) that is not less than (i.e., greater than or equal to) \(\text{value}\), or \(\text{last}\) if no such element is found
  - **Complexity**: the number of comparisons performed is logarithmic in the distance between \(\text{first}\) and \(\text{last}\) (at most \(\log_2(\text{last} - \text{first}) + O(1)\) comparisons)

For non-random-access iterators, the number of iterator increments is linear

```cpp
template<class ForwardIter, class T>
ForwardIt lower_bound(ForwardIter first, ForwardIter last, const T& value);
```
Algorithms – A Sample remove_copy_if

- remove_copy_if
  - **Action**: copies elements from the range \([\text{first}, \text{last})\), to another range beginning at \(\text{dest}\), omitting the elements which satisfy specific criteria; source and destination ranges cannot overlap; returns an iterator to the element past the last element copied.
  - **Complexity**: exactly \(\text{std::distance}(\text{first}, \text{last})\) applications of the predicate.

```cpp
template<class InputIter, class OutputIter, class UnaryPredicate>
OutputIter
remove_copy_if(InputIter first, InputIter last, OutputIter dest, UnaryPredicate pred) {
    for (; first != last; ++first) {
        if (!pred(*first)) {
            *dest++ = *first;
        }
    }
    return dest;
}
```
Summary
Summary: Key Principles

- **Containers** store collections of *elements*

- **Algorithms** perform operations upon collections of elements

- Containers and algorithms are entirely independent

- **Iterators** provide a common unit of information exchange between containers and algorithms

- STL makes complexity guarantees by specifying *interfaces* and *requirements*
Summary: On the Brilliance of the STL

• Four important positive qualities
  • Speed
  • Efficiency
  • Extensibility
  • Elegance

• The STL separates data structures from algorithms, and ties them together with iterators
  • It is remarkable can be done with only 5 iterator categories

• The underlying ideas have become embedded into our way of thinking
Thank You for Attending!

Talk:  github.com/BobSteagall/CppCon2021
Blog:  bobsteagall.com