Tecton Description of STL Container and Iterator Concepts

David R. Musser[*]
Computer Science Department
Rensselaer Polytechnic Institute, Troy, NY 12180
musser@cs.rpi.edu

Incomplete Draft, November 3, 1998

Abstract

The various concepts of containers and iterators used in the C++ Standard Template Library (STL) are stated as Tecton concept descriptions.

[*]This work was performed while the author was on sabbatical at the Wilhelm-Schickard-Institut für Informatik, Universität Tübingen.

1 Introduction

The purpose of this note is to develop formal specifications of the C++ Standard Template Library (STL) [3, 4] container and iterator concepts, expressing them in the Tecton language [1]. In general our starting point is the set of semi-formal concept descriptions given in [6], since that is already organized as a concept hierarchy and is more complete in its statement of semantics than the actual C++ standard [5]. [[So far there are some places where we have to depart from [6], as noted. Or maybe not properly noted yet, but they will be.]]

2 Overview of Concept Definitions and Lemmas

"container.tec" 1 ≡
⟨Basic concepts 2a⟩
⟨Basic-container concept 6a⟩
⟨Trivial iterator concept 6b⟩
⟨Input iterator concept 7a⟩
⟨Output iterator concept 7b⟩
⟨Container concept 8a⟩
⟨Forward iterator concept 8b⟩
⟨Extension of forward iterator concept 8c⟩
⟨Constness extensions of forward iterator concept 9a⟩
⟨Forward container concept 9c⟩
⟨Range equivalence concept 10a⟩
⟨Access equivalence concept 10b⟩
⟨Lexicographical order concept 9f⟩
⟨Indexing concepts 11a⟩
⟨Basic sequence concept 11b⟩
⟨Sequence concept 13a⟩
⟨Default construction lemma 12c⟩
⟨Front insertion sequence concept 14a⟩
⟨Back insertion sequence concept 14b⟩
⟨Bidirectional iterator concept 14c⟩
⟨Random-access iterator concept 15a⟩
⟨Reversible container concept 15b⟩
⟨Some invariants of reversible containers 15c⟩
⟨Random access container concept 16a⟩
⟨Vector concept 16b⟩
⟨Deque concept 16c⟩
⟨Spliceable container concept 17a⟩
⟨List concept 17b⟩
⟨Associative container concept 20a⟩
⟨Sorted associative container concept 20b⟩
⟨Hashed associative container concept 20c⟩
⟨Multiple associative container concept 21a⟩
⟨Unique associative container concept 21b⟩
⟨Simple associative container concept 21c⟩
⟨Pair and associative-pair concepts 21d⟩
⟨Pair associative container concept 22a⟩
⟨Combinations of associative container concepts 22b⟩
◇

3 Basic Concepts

There are a few basic concepts needed in preparation for defining the general container concept.

⟨Basic concepts 2a⟩ ≡
⟨Boolean and Natural concepts 2b⟩
⟨Binary relation concepts 3⟩
⟨Total order concept 4a⟩
⟨General binary relation concept 4b⟩
⟨Function set concept 4c⟩
⟨Object, accessible object, and writable object concepts 5a⟩
⟨Assignable object concept 5b⟩
⟨Default constructible object concept 5c⟩
⟨Equality-comparable concept 5d⟩
⟨Strict weak order concept 9d⟩
⟨Less-than-comparable concept 9e⟩
◇

Used in part 1.

The following concept descriptions come from [2].

⟨Boolean and Natural concepts 2b⟩ ≡

```
Definition: Boolean
  introduces bool,
     true -> bool,
     false -> bool;
  generates bool freely using true, false.

Remark: The above concept is actually predefined by the semantics.

Definition: Set
    uses Boolean;
    introduces domain.

Definition: Natural
    refines Set [with naturals as domain];
    introduces 0 -> naturals,        1 -> naturals,
       next(naturals) -> naturals,   +(naturals, naturals) -> naturals,
       -(naturals, naturals) -> naturals,
       *(naturals, naturals) -> naturals;
    generates naturals freely using 0, next;
    requires (for n, m: naturals)
       n + 0 = n,                       n + next(m) = next(n + m),
       n - 0 = n,                       0 - n = 0,
       next(n) - next(m) = n - m,       1 = next(0),
       i * 0 = 0,                       i * next(j) = i * j + i.
Precedence: {<, <=, >=, >, =}.

Extension:  Natural
    introduces <=(naturals, naturals) -> bool;
    requires (for n, m: naturals)
       0 <= n,                        not(next(n) <= 0),
       (next(m) <= next(n)) = (m <= n).

Precedence: {implies} < {or, xor} < {and} < {=}.

Extension: Boolean
  introduces
      not(bool) -> bool,
      and(bool, bool) -> bool,
      or(bool, bool) -> bool,
      xor(bool, bool) -> bool,
      implies(bool, bool) -> bool;
```

```
  requires (for x, y: bool)
      not(true) = false,
      not(false) = true,
      (true and x) = x,
      (false and x) = false,
      (x or y) = not(not(x) and not(y)),
      (x xor y) = not(x = y),
      (x implies y) = (not(x) or y).
◇
```

Used in part 2a.

The following concept descriptions also come from [2].

⟨Binary relation concepts 3⟩ ≡

```
Definition: Binary-relation
  uses Set;
  introduces R(domain, domain) -> bool.

Precedence: {R, =}.

Definition: Reflexive
  refines Binary-relation;
  requires (for x: domain) x R x.

Definition: Anti-reflexive
  refines Binary-relation;
  requires (for x: domain) not(x R x).

Definition: Symmetric
  refines Binary-relation;
  requires (for x, y: domain) x R y implies y R x.

Definition: Anti-symmetric
  refines Binary-relation;
  requires (for x, y: domain) x R y and y R x implies x = y.

Definition: Transitive
  refines Binary-relation;
  requires (for x, y, z: domain) x R y and y R z implies x R z.

Definition: Equivalence-relation
  refines Reflexive, Symmetric, Transitive.

Definition: Partial-order
  refines Reflexive [with <= as R],
          Anti-symmetric [with <= as R],
          Transitive [with <= as R].

Definition: Strict-partial-order
  refines Anti-reflexive [with < as R],
          Transitive [with < as R].

Extension: Partial-order
   introduces <(x: domain, y: domain) = ((x <= y) and not(x = y)).

Lemma: Partial-order implies Strict-partial-order.
◇
```

Used in part 2a.

Note that the properties of Partial-order are stated in terms of `<=` while those of Strict-partial-order are in terms of `<`.

⟨Total order concept 4a⟩ ≡

```
Definition: Total-order
   refines Partial-order;
   requires (for x, y: domain) x <= y or y <= x.
◇
```

Used in part 2a.

For some purposes we need binary relations on two distinct domains.

⟨General binary relation concept 4b⟩ ≡

```
Definition: General-binary-relation
uses Set [with domain1 as domain], Set [with domain2 as domain];
introduces R(domain1, domain2) -> bool.

Lemma: Binary-relation is General-binary-relation.
◇
```

Used in part 2a.

[[Or we could have introduced General-binary-relation first and then written

```
Definition: Binary-relation refines
   General-binary-relation [with domain as domain1, domain as domain2].
```

]]

⟨Function set concept 4c⟩ ≡

```
Abbreviation: Range
is Set [with codomain as domain].

Definition: Function-set
refines Set [with functions as domain];
uses Set, Range;
introduces value(functions, domain) -> codomain.

Definition: Binary-function-set
refines Set [with functions as domain];
uses Set [with domain1 as domain], Set [with domain2 as domain], Range;
introduces value(functions, domain1, domain2) -> codomain.

Abbreviation: Predicate-set
is Function-set [with predicates as functions, bool as codomain].

Abbreviation: Binary-predicate-set
is Function-set [with binary-predicates as functions, bool as codomain].
◇
```

Used in part 2a.

We now define some of the most basic concepts needed for developing container concepts.

⟨Object, accessible object, and writable object concepts 5a⟩ ≡

```
Abbreviation: Object is Set [with objects as domain].

Abbreviation: Element is Set [with elements as domain].

Definition: Accessible refines Object;
uses Element;
introduces access(objects) -> elements.

Definition: Writable refines Object;
uses Element;
introduces :=(objects, elements) -> objects,
  constant(objects) -> bool.
◇
```

Used in part 2a.

⟨Assignable object concept 5b⟩ ≡

```
Definition: Assignable refines Accessible, Writable;
introduces
   copy(objects) -> objects,
   swap(objects, objects) -> (objects, objects);
requires (for x, y: objects; z: objects; e: elements)
   access(x := e) = e,
   access(copy(z)) = access(z),
   swap(x, y) = (y, x).
◇
```

Used in part 2a.

Note: The above definition uses a cartesian product as a range type (of `swap`), which wasn't allowed in the original Tecton language definition [1], but which seems like a useful extension (suggested by R. Loos). I've also started using "requires" instead of "obeys" in definitions (also suggested by R. Loos). I think it may be better though to stick with "obeys" in lemmas, since those properties are not additional requirements but rather additional properties that follow from the axioms.

⟨Default constructible object concept 5c⟩ ≡

```
Definition: Default-constructible refines Accessible;
introduces construct() -> objects.
◇
```

Used in part 2a.

⟨Equality-comparable concept 5d⟩ ≡

```
Definition: Equality-comparable
  refines Object;
  introduces  =(objects, objects) -> bool,
             /=(objects, objects) -> bool.
◇
```

Used in part 2a.

[[There may be a problem here with the built-in semantics of `=` being stronger than what is supposed to be required in STL concepts. May need to use another identifier instead of =.
]

4 Container and Iterator Concepts

Initially we define a basic container concept very simply, but after defining the input iterator concept we define the container concept with operations that depend on iterators.

⟨Basic-container concept 6a⟩ ≡
```
Definition: Basic-container
refines Assignable [with containers as objects];
uses Accessible.
◇
```
Used in part 1.

We depart here from [6], both in first defining a Basic-container concept before defining Container, and in requiring the objects in the container to be only Accessible rather than Assignable. The first difference is due only to technical requirements of Tecton, but the second is necessary because [6] appears to be in error in requiring Assignable for the objects. The error becomes apparent when the Associative Container concept is introduced in [6] as a refinement of Container but with the statement that the objects in an associative container are not necessarily assignable. (For containers other than associative containers, such as Forward-container, Sequence, etc., we obtain the assignable property for contained objects as a combination of the accessible and writable properties as introduced via the Forward-iterator concept below.)

Although it is a unary operator on C++ pointers and on their STL generalization, iterators, the dereferencing operator, `*`, must be expressed here as a binary operator depending on the container as well as the iterator.

⟨Trivial iterator concept 6b⟩ ≡
```
Definition: Trivial-iterator
refines Assignable [with iterators as objects],
        Default-constructible [with iterators as objects],
        Equality-comparable [with iterators as objects];
uses Accessible, Basic-container;
introduces *(containers, iterators) -> objects.
◇
```
Used in part 1.

Note that the `objects` sort used as the range of `*` is from the Accessible concept. So trivial iterators give a means of obtaining accessible, but not necessarily assignable, objects.

In defining input iterators the main difficulty is to avoid making overly strong assertions about the meaning of `++`, since we want to allow models in which not only the iterator but also the container into which it points is changed by the operation. Thus we make the range of `++` the cartesian product of `containers` and `iterators` rather than just `iterators`. As with the dereferencing operator, `++` is a unary operator on C++ pointers and on their STL generalization, iterators, but we must express it here as a binary operator depending on the container as well as the iterator.

⟨Input iterator concept 7a⟩ ≡

```
Definition: Input-iterator refines Trivial-iterator;
uses Natural, Basic-container;
introduces ranges,
  ++(containers, iterators) -> (containers, iterators),
  range(containers, iterators, iterators) -> ranges,
  in(iterators, ranges) -> bool,
  next(containers, iterators, naturals) -> iterators (private),
  valid(ranges, naturals) -> bool,
  size(ranges) -> naturals;
requires (i, j, p, q: iterators; c, d: containers; n: naturals)
  next(c, i, 0) = i,
  next(c, i, n + 1) = next(d, p, n) where c++i = (d, p),
  valid(range(c, i, j)) = ((for some k: naturals) next(c, i, k) = j),
  valid(range(c, i, j)) and valid(range(c, p, q)) implies
   (range(c, i, j) = range(c, p, q)) =
     if i = j then p = q
       else i = p
            and range(c1, i1, j) = range(c1, i1, q)
            where c++i = (c1, i1).
  valid(range(c, i, j)) implies
    (p in range(c, i, j)) = (i /= j and (p = i or p in range(c1, i1, j))
                            where c++i = (c1, i1),
  valid(range(c, i, j)) implies
    size(range(c, i, j)) = if i = j then 0
                        else size(range(d, p, j)) where c++i = (d, p).
◇
```

Used in part 1.

The output iterator concept gives us the ability to assign to the objects to which the iterators point, but not necessarily to access them.

⟨Output iterator concept 7b⟩ ≡

```
Definition: Output-iterator
refines Assignable [with iterators as objects],
        Default-constructible [with iterators as objects];
uses Writable, Basic-container;
introduces
  *(containers, iterators) -> objects,
  ++(containers, iterators) -> (containers, iterators).
◇
```

Used in part 1.

We can now define the container concept:

⟨Container concept 8a⟩ ≡

```
Definition: Container
refines Basic-container;
uses Input-iterator;
introduces
  nonempty-containers < containers,
  begin(containers) -> iterators,
  end(containers) -> iterators,
  size(containers) -> naturals,
  empty(containers) -> bool;
requires (for x: containers)
  x: nonempty-containers = not(empty(x)),
  size(x) = size(range(x, begin(x), end(x))),
  empty(x) = (size(x) = 0),
  valid(range(x, begin(x), end(x))).
```

◇

Used in part 1.

5 Forward Iterator and Forward Container Concepts

In the forward iterator concept the `++` function is required to leave the container unchanged.

⟨Forward iterator concept 8b⟩ ≡

```
Definition: Forward-iterator
refines Input-iterator, Output-iterator;
introduces ++(containers, iterators) -> iterators;
requires (for i, p: iterators; c, c1: containers)
  c++i = (c1, p) implies c1 = c,
  c++i = p where c++i = (c, p).
```

◇

Used in part 1.

The above formulation depends on being able to overload function identifiers and resolve them where there is no difference in the argument types, only in the range type. [[If this assumption causes too much trouble, we could avoid it by using some other identifier in place of `++` in the input iterator and output iterator concepts.]]

Note that Input-iterator uses Accessible and Output-iterator uses Writable (in both cases giving properties to the objects to which the iterators point), and both of these concepts together imply Assignable. Since Forward-iterator refines both Input-iterator and Output-iterator, we may treat Assignable as a formal parameter of Forward-iterator. This fact may be stated explicitly in Tecton as an extension:

⟨Extension of forward iterator concept 8c⟩ ≡

```
Extension: Forward-iterator uses Assignable.
```

◇

Used in part 1.

The legality of this extension depends on the above mentioned relationships.

We also extend Forward-iterator to have some predicates relating to "constness." These predicates are not fully defined here, but they do obey a constraint among themselves.

⟨Constness extensions of forward iterator concept 9a⟩ ≡

```
Extension: Forward-iterator
introduces
  constant(containers) -> bool,
  constant-in-object-type(containers) -> bool,
  points-to-constant(iterators) -> bool;
requires (for c: containers; i: iterators)
  constant(c*i) = (constant-in-object-type(c) or constant(c)
                  or points-to-constant(i)).
◇
```

Used in part 1.

These predicates can be used to model C++ declarations involving the `const` attribute and `const_iterators`.

Another observation is that with forward iterators the valid-range predicate is transitive.

⟨Transitivity of valid-range 9b⟩ ≡

```
Lemma: Forward-iterator
obeys (for i, j, k: iterators; c: containers)
  valid(range(c, i, j)) and valid(range(c, j, k))
    implies valid(range(c, i, k)).
◇
```

Not used.

A transitity property could be stated for even input iterators, but it would be more complicated (and weaker) because the container can be modified by `++`.

The forward container concept is now defined in terms of the container concept by merely substituting the forward iterator concept for that of input iterators.

⟨Forward container concept 9c⟩ ≡

```
Definition: Forward-container
refines Container [with Forward-iterator as Input-iterator].
◇
```

Used in part 1.

When there is an ordering requirement on the object concept, we have an ordering on forward containers. Frequently this requirement is stated as a strict total order, but the weaker notion of *strict weak order* suffices.

⟨Strict weak order concept 9d⟩ ≡

```
Definition: Strict-weak-order
refines Strict-partial-order, Transitive [with equivalent as R];
requires (for x, y, z: domain)
    equivalent(x, y) = (not(x < y) and not(y < x)).
◇
```

Used in part 2a.

⟨Less-than-comparable concept 9e⟩ ≡

```
Abbreviation: Less-than-comparable
  is Strict-weak-order.
◇
```

Used in part 2a.

⟨Lexicographical order concept 9f⟩ ≡

```
Definition: Lexicographical-order
uses Forward-iterator,
     Strict-weak-order [with Object as Set, objects as domain];
introduces
```

```
  less-than(containers, containers,iterators, iterators) -> bool;
requires (for c, d: containers; i, j, k, l: iterators)
  less-than(c, d, i, i, k, k) = false,
  k /= l implies less-than(c, d, i, i, k, l),
  valid(range(c, i, j)) and i /= j implies
     less-than(c, d, i, j, k, k) = false,
  valid(range(c, i, j)) and i /= j and valid(range(d, k, l))
     and k /= l implies
       less-than(c, d, i, j, k, l) =
        (c*i < d*k) or
         c*i = d*k and less-than(c, d, c++i, j, d++k, l))).

Definition: Forward-container-with-less-than
refines Forward-container [with Strict-weak-order as ];
uses Lexicographical-order;
introduces <(containers, containers) -> bool;
requires (for x, y: containers)
  (x < y) = less-than(x, y, begin(x), end(x), begin(y), end(y)).
◇
```
Used in part 1.

If objects of a container have an equivalence relation, `equivalent-to`, defined on them, we define equivalence of ranges of iterators as follows:

⟨Range equivalence concept 10a⟩ ≡
```
Definition: Range-equivalence
uses Forward-iterator,
     Equivalence-relation [with Object as Set, objects as domain,
                                equivalent-to as R];
introduces
  equivalent-range(containers, containers, iterators, iterators, iterators)
     -> bool;
requires (for c, d: containers; i, j, k: iterators)
  equivalent-range(c, d, i, i, k),
  valid(range(c, i, j)) and i /= j implies
     equivalent-range(c, d, i, j, k) =
       (c*i equivalent-to d*k
        and equivalent-range(c, d, c++i, j, d++k)).

Definition: Forward-container-with-equivalence
refines Forward-container;
uses Range-equivalence;
introduces equivalent-to(containers, containers) -> bool;
requires (for x, y: containers)
  (x equivalent-to y) =
   (size(x) = size(y) and
      equivalent-range(x, y, begin(x), end(x), begin(y))).
◇
```
Used in part 1.

A fundamental kind of equivalence that is used with the above definition is access equivalence:

⟨Access equivalence concept 10b⟩ ≡
```
Definition: Access-equivalence
refines Equivalence-relation [Accessible as Set, with objects as domain,
                              access-equivalent-to as R];
requires (for x, y: objects)
  (x access-equivalent-to y) = (access(x) = access(y)).
◇
```
Used in part 1.

6 Sequence Concept

In the sequence concept we strengthen the container requirements sufficiently to guarantee the elements are maintained in a fixed order, obtainable (repeatably) by iterating through the container. Constructors and insertion and deletion functions are provided, and their use determines the order in which the elements appear. In order to specify the order precisely, we first introduce several indexing-related concepts.

⟨Indexing concepts 11a⟩ ≡
```
Abbreviation: Index-set is Set [with indices as domain].

Definition: Indexed-container refines Container; uses Index-set;
introduces access(nonempty-containers, indices) -> elements.

Definition: Least-in-partial-order refines Partial-order;
introduces least -> domain;
requires (for x: domain) least <= x.

Definition: Enumerative-total-order
refines Least-in-partial-order, Total-order;
uses Natural;
introduces ranges,
     next(domain) -> domain,
     +(domain, naturals) -> domain,
     -(domain, domain) -> naturals,
     range(domain, domain) -> ranges,
     in(domain, ranges) -> bool;
generates domain freely using least, next;
requires (for x, y, z: domain; n: naturals)
     x < next(x),
     x + 0 = x,
     x + next(n) = next(x + n),
     y = x + n implies y - x = n,
     x in range(y, y) = false,
     y < z implies (x in range(y, z)) = (x = y or (x in range(next(y), z))).
```
◇

Used in part 1.

The sequence concept introduces several new functions, including two versions each of insertion and deletion, which have relatively complex semantics. Here we first introduce a basic sequence concept without insertion operations, so that it can also be used in the definition of associative containers (which have a different notion of insertion from that of sequences).

⟨Basic sequence concept 11b⟩ ≡
```
Definition: Basic-sequence
refines
  Forward-container-with-equivalence
     [with Access-equivalence as Equivalence-relation,
           access-equivalent-to as equivalent-to,
           sequences as containers],
Default-constructible [with containers as objects],
  Indexed-container [with Enumerative-total-order as Index-set,
                          indices as domain,
                          sequences as containers,
                          nonempty-sequences as nonempty-containers,
                          Default-constructible as Element,
                          construct-element as construct];
introduces
  position(containers, iterators) -> indices,
```

```
    construct(naturals, elements) -> sequences,
    construct(naturals) -> sequences,
    construct(containers, iterators, iterators) -> sequences,
    erase(nonempty-sequences, iterators) -> sequences,
    erase(nonempty-sequences, iterators, iterators) -> sequences;
  requires (for c, c1, c2: sequences; d, d1: nonempty-sequences;
            i, i1, j, k: iterators; e: elements; p, q: indices; n: naturals)
```

⟨Axioms for position function 12a⟩
⟨Axioms for construct functions 12b⟩
⟨Axioms for erase functions 12d⟩
◇

Used in part 1.

The position function is used to define the positions in the container relative to the first position, which is assigned the index `least` defined in the Least-in-partial-order concept.

⟨Axioms for position function 12a⟩ ≡

```
  position(c, begin(c)) = least,
  position(c, i) = p and p < least + size(c) implies position(c, c++i) = next(p),
  position(d, i) < least + size(d) implies
      access(d*i) = access(d, position(d, i)),
```

◇

Used in part 11b.

The `access` function on the left of the above equation is from the Accessible concept, while the one on the right is from the Indexed-container concept. Note that this axiom is written using `d`, an identifier declared to be of the `nonempty-sequence` sort.

⟨Axioms for construct functions 12b⟩ ≡

```
  end(construct()) = begin(construct()),
  c1 = construct(n, e) implies
     size(c1) = n and
      ((for p: indices) p in range(least, least + size(c1))
          implies access(c1, p) = e),
  construct(n) = construct(n, construct-element()),
  c1 = construct(c, j, k) implies
     size(c1) = size(range(c, j, k)) and
      ((for p: indices) p in range(least, least + size(c1))
           implies access(c1, p) = access(c, position(c, j) + (p - least)),
```

◇

Used in part 11b.

Another way to specify the last construct function would be in terms of the corresponding insert function.

As a trivial consequence of the axiom about `construct()` and the `size` axioms we have the following

⟨Default construction lemma 12c⟩ ≡

```
  Lemma: Sequence obeys size(construct()) = 0.
```

◇

Used in part 1.

⟨Axioms for erase functions 12d⟩ ≡

```
  i in range(d, begin(d), end(d)) and c1 = erase(d, i) implies
     size(c1) = size(d) - 1 and
     ((for p, q: indices)
        (p in range(least, position(d, i))
           implies access(c1, p) = access(d, p)) and
```

```
      (q in range(position(d, i), least + size(d))
         implies access(c1, q) = access(d, next(q)))),
i in range(d, begin(d), end(d)) and j in range(d, begin(d), end(d))
  and position(d, i) < position(d, j) and size(d) >= size(range(d, i, j))
  and c1 = erase(d, i, j)
   implies
     size(c1) = size(d) - size(range(d, i, j))
     and ((for p, q: indices)
          (p in range(least, position(d, i))
             implies access(c1, p) = access(d, p))
          and (q in range(position(d, i), least + size(d))
            implies access(c1, q) = access(d, q + size(range(d, i, j))))).
◇
```

Used in part 11b.

The sequence concept is a refinement of the basic sequence concept with functions for inserting elements and getting the front element.

⟨Sequence concept 13a⟩ ≡

```
Definition: Sequence
refines Basic-sequence;
introduces
  insert(sequences, iterators, elements) -> (sequences, iterators),
  insert(sequences, sequences, iterators, iterators) -> sequences,
  front(nonempty-sequences) -> elements;
requires (for c, c1, c2: sequences; d, d1: nonempty-sequences;
          i, i1, j, k: iterators; e: elements; p, q: indices; n: naturals)
  ⟨Axiom for inserting one element 13b⟩
  ⟨Axiom for inserting a range of elements 13c⟩
  front(d) = d*begin(d).
◇
```

Used in part 1.

⟨Axiom for inserting one element 13b⟩ ≡

```
(c1, i1) = insert(c, i, e) implies
   size(c1) = size(c) + 1 and access(c1*i1) = e
   and position(c1, i1) = position(c, i)
   and ((for p, q: indices)
         (p in range(least, position(c, i))
             implies access(c1, p) = access(c, p))
         and (q in range(position(c, i), least + size(c))
             implies access(c1, next(q)) = access(c, q))),
◇
```

Used in part 13a.

⟨Axiom for inserting a range of elements 13c⟩ ≡

```
c2 = insert(c, i, c1, j, k) implies
   size(c2) = size(c) + size(range(c1, j, k))
   and ((for p: indices)
         p in range(position(c, i), position(c, i) + size(range(c1, i, j)))
         implies access(c2, p) = access(c1, position(c, j) + (p - least)))
   and ((for p, q: indices)
         (p in range(least, position(c, i))
             implies access(c2, p) = access(c, p))
         and (q in range(position(c, i), least + size(c))
             implies access(c1, q + size(range(c1, j, k))) = access(c, q))),
◇
```

Used in part 13a.

7 Front Insertion Sequence Concept

⟨Front insertion sequence concept 14a⟩ ≡

```
Definition: Front-insertion-sequence
refines Sequence;
introduces
  push-front(sequences, objects) -> sequences,
  pop-front(nonempty-sequences) -> sequences;
requires (for s, s1: sequences; x: objects)
  access(front(push-front(s, x))) = access(x),
  pop-front(push-front(s, x)) access-equivalent-to s.
◇
```

Used in part 1.

8 Back Insertion Sequence Concept

⟨Back insertion sequence concept 14b⟩ ≡

```
Definition: Back-insertion-sequence refines Sequence;
introduces
  back(nonempty-sequences) -> objects,
  push-back(sequences, objects) -> sequences,
  pop-back(nonempty-sequences) -> sequences;
requires (for s: sequences; s1: nonempty-sequences; x: objects; i: iterators)
  back(s1) = s1*i where s1++i = end(s1),
  access(back(push-back(s, x))) = access(x),
  pop-back(push-back(s, x)) access-equivalent-to s.
◇
```

Used in part 1.

9 Bidirectional Iterator Concept

⟨Bidirectional iterator concept 14c⟩ ≡

```
Definition: Bidirectional-iterator refines Forward-iterator;
introduces --(containers, iterators) -> iterators;
requires (for d: nonempty-containers; i: iterators)
  d++j = i implies d--i = j.
◇
```

Used in part 1.

10 Random Access Iterator Concept

⟨Random-access iterator concept 15a⟩ ≡

```
Definition: Random-access-iterator
refines Bidirectional-iterator, Less-than-comparable;
introduces +(containers, iterators, naturals) -> iterators,
  -(containers, iterators, naturals) -> iterators,
  brackets(containers, iterators, naturals) -> objects;
requires (for c: containers; d: nonempty-containers; i, j: iterators;
          n: Naturals)
  i in range(c, begin(c), end(c)) and size(range(c, i, end(c)) >= n implies
     +(c, i, n) = if n = 0 then i else +(c, c++i, n-1),
  i in range(c, begin(c), end(c)) and size(range(c, begin(c), i) >= n implies
     -(c, i, n) = if n = 0 then i else -(c, c--i, n-1),
  i in range(c, begin(c), end(c)) and i + n < end(d) implies
     brackets(d, i, n) = d*(+(d, i, n)),
  i in range(c, begin(c), end(c)) and j in range(c, begin(c), end(c)) implies
    i < j = ((for some n: Naturals) i + n = j).
◇
```

Used in part 1.

[[Having to have the container as an argument makes + and - ternary operations and thus infix notation is ruled out, which is an inconvenience.]]

11 Reversible Container Concept

⟨Reversible container concept 15b⟩ ≡

```
Definition: Reversible-container
refines Forward-container [with Bidirectional-iterator as Forward-iterator];
uses Bidirectional-iterator [with reverse-iterators as iterators];
introduces
  reverse-iterator(iterators) -> reverse-iterators,
  rbegin(containers) -> reverse-iterators,
  rend(containers) -> reverse-iterators;
requires (for c: containers; i: iterators; r: reverse-iterators)
  rbegin(c) = reverse-iterator(end(c)),
  rend(c) = reverse-iterator(begin(c)),
  r = reverse-iterator(i) implies c++r = c--i and c--r = c++i.
◇
```

Used in part 1.

Note that `c++r` and `c--r` are legal because of the `uses` clause.

⟨Some invariants of reversible containers 15c⟩ ≡

```
Lemma: Reversible-container
obeys (for c: containers)
  valid(range(c, rbegin(c), rend(c))),
  size(range(c, rbegin(c), rend(c)) = size(c).
◇
```

Used in part 1.

12 Random Access Container Concept

⟨Random access container concept 16a⟩ ≡

```
Definition: Random-access-container
refines Reversible-container [with Random-access-iterator as
                                       Bidirectional-iterator];
introduces brackets(containers, naturals) -> objects;
requires (d: nonempty-containers; n: naturals)
  n < size(d) implies brackets(d, n) = d*(+(d, begin(d), n)).
◇
```

Used in part 1.

13 Vector Concept

⟨Vector concept 16b⟩ ≡

```
Definition: Vector
refines Random-access-container [with vectors as containers,
                                     nonempty-vectors as nonempty-containers],
        Back-insertion-sequence [with vectors as sequences,
                                     nonempty-vectors as nonempty-sequences];
introduces capacity(vectors) -> naturals,
  reserve(vectors, naturals) -> vectors,
  usable(vectors, iterators) -> bool;
generates vectors freely using construct, push-back;
requires (for v, v1: vectors; w: nonempty-vectors; n: naturals;
              i, i1, j, k: iterators; e: elements)
  capacity(v) >= size(c),
  capacity(reserve(v, n)) >= n,
  v1 = reserve(v, n) implies
     size(v1) = size(v)
     and v1 access-equivalent-to v,
  n <= capacity(v) implies reserve(v, n) = v,
  usable(v, begin(v)),
  usable(v, end(v)),
  valid(range(v, i, j)) and
    usable(v, i) and usable(v, j) and k in range(v, i, j) implies usable(v, k),
  i in range(w, begin(w), end(w)) implies not(usable(erase(w, i), end(w))),
  i in range(w, begin(w), end(w)) and j in range(w, begin(w), end(w))
    and position(w, i) < position(w, j) and n = size(range(w, i, j))
    and k in range(w, end(w) - n, end(w)) implies
      not(usable(erase(w, i, j), k)).
◇
```

Used in part 1.

14 Deque Concept

The deque concept simply combines three of the previously given concepts.

⟨Deque concept 16c⟩ ≡

```
Definition: Deque
refines Random-access-container [with deques as containers,
                                     nonempty-deques as nonempty-containers],
        Front-insertion-sequence [with deques as sequences,
                                     nonempty-deques as nonempty-sequences],
        Back-insertion-sequence [with deques as sequences,
                                     nonempty-deques as nonempty-sequences];
generates deques freely using construct, push-back.
◇
```

Used in part 1.

15 List Concept

We first introduce a splice operation on iterator ranges:

⟨Spliceable container concept 17a⟩ ≡

```
Definition: Spliceable-container
refines Reversible-container;
introduces lists,
  +(ranges, ranges) -> ranges,
  splice(lists, iterators, lists) -> lists,
  splice(lists, iterators, lists, iterators) -> lists,
  splice(lists, iterators, lists, iterators, iterators) -> lists;
requires (for c, d, e: containers; i, j, k, l, p, q: iterators)
  range(e, p, q) = range(c, i, j) + range(d, k, l)
    implies if i = j then e = d and p = k and q = l
            else range(e, p, j) = range(c, i, j) and e++(c--j) = k
                 and range(e, k, l) = range(d, k, l),
  w = splice(u, p, v) implies
    range(w, begin(w), end(w)) =
      range(u, begin(u), p) + range(v, begin(v), end(v)) + range(u, p, end(u)),
  w = splice(u, p, v, i) implies
    range(w, begin(w), end(w)) =
      range(u, begin(u), p) + range(v, i, v++i) + range(u, p, end(u)),
  w = splice(u, p, v, i, j) implies
    range(w, begin(w), end(w)) =
      range(u, begin(u), p) + range(v, i, j) + range(u, p, end(u)).
◇
```

Used in part 1.

[[The following definition should be broken down into smaller parts for easier understanding.]]

⟨List concept 17b⟩ ≡

```
Definition: List
refines Front-insertion-sequence [with lists as sequences,
                                  nonempty-lists as nonempty-sequences],
  Back-insertion-sequence [with lists as sequences,
                           nonempty-lists as nonempty-sequences],
  Spliceable-container [with lists as containers,
                        nonempty-lists as nonempty-containers];
  uses Predicate-set [with elements as domain],
       Binary-predicate-set [with elements as domain];
introduces
  remove(lists, elements) -> lists,
  remove(lists, iterators, iterators, elements) -> lists (private),
  remove-if(lists, predicates) -> lists,
  remove-if(lists, iterators, iterators, predicates) -> lists (private),
  unique(lists) -> lists,
  unique-if(lists, predicates) -> lists,
  interleaving(containers, containers, containers) -> bool (private),
  interleaving(containers, iterators, iterators,
               containers, iterators, iterators,
               containers, iterators, iterators) -> bool (private),
  ordered(containers, binary-predicates) -> bool (private),
  ordered(containers, iterators, iterators, binary-predicates) -> bool (private),
  merge(lists, lists) -> lists,
  merge(lists, lists, binary-predicates) -> lists,
  count(containers, elements) -> naturals (private),
  count(containers, iterators, iterators, elements) -> naturals (private),
```

```
  sort(lists) -> lists,
  sort(lists, binary-predicates) -> lists,
  reverse(lists) -> lists,
  reverse(lists, iterators, iterators) -> lists (private);
generates lists freely using construct, push-back;
requires (for u, v, w: lists; e: elements; i, j, p, q, r: iterators;
              b: predicates; c: binary-predicates)
  i in range(u, begin(u), end(u)) and w = erase(u, i) implies
    range(w, begin(w), end(w)) =
       range(u, begin(u), i) + range(u, u++i, end(u)),
  remove(u, e) = remove(u, begin(u), end(u), e),
  valid(range(u, i, j)) implies
    remove(u, i, j, e) = if i = j then u
                         else if access(u*i) = e then
                            remove(erase(u, i), u++i, j, e)
                         else
                            remove(u, u++i, j, e),
  remove-if(u, b) = remove-if(u, begin(u), end(u), b),
  valid(range(u, i, j)) implies
    remove-if(u, i, j, b) = if i = j then u
                         else if value(b, access(u*i)) then
                            remove-if(erase(u, i), u++i, j, b)
                         else
                            remove-if(u, u++i, j, b),
  unique(u) = unique(u, begin(u), end(u)),
  valid(range(u, i, j)) implies
    unique(u, i, j) = if i = j then u else
                      if u++i = j then u else
                      if access(u*i) = access(u*(u++i))
                        then unique(erase(u, i), u++i, j)
                        else unique(u, u++i, j),
  unique-if(u, b) = unique-if(u, begin(u), end(u), b),
  valid(range(u, i, j)) implies
    unique-if(u, i, j, b) = if i = j then u else
                      if u++i = j then u else
                      if value(b, access(u*i), access(u*(u++i)))
                        then unique-if(erase(u, i), u++i, j, b)
                        else unique-if(u, u++i, j, b),
  reverse(u) = reverse(u, begin(u), end(u)),
  valid(range(u, i, j)) implies
    reverse(u, i, j) = if i = j then u
                       else splice(reverse(u, u++i, j), j, u, i),
  interleaving(u, v, w) = interleaving(u, begin(u), end(u),
                                       v, begin(v), end(v),
                                       w, begin(w), end(w)),
  valid(range(u, i, j)) and valid(range(v, p, q))
    and valid(range(w, r, s)) implies
    interleaving(u, i, j, v, p, q, w, r, s) =
      if i = j then
         if p = q then r = s else r /= s and v*p = w*r and
                                 interleaving(u, i, j, v, v++p, q, w, w++r, s)
      else if p = q then r /= s and u*i = w*r and
                                interleaving(u, u++i, j, v, p, q, w, w++r, s)
      else r /= s and (u*i = w*r and
                         interleaving(u, u++i, j, v, p, q, w, w++r, s)
                       or v*p = w*r and
                            interleaving(u, i, j, v, v++p, q, w, w++r, s)),
  ordered(u) = ordered(u, begin(u), end(u)),
  valid(range(u, i, j)) implies
```

```
  ordered(u, i, j) =
     (i = j or u++i = j or not(access(u*(u++i)) < access(u*i))
                           and ordered(u, u++i, j)),
ordered(u, c) = ordered(u, begin(u), end(u), c),
valid(range(u, i, j)) implies
  ordered(u, i, j, c) =
     (i = j or u++i = j or not(value(c, access(u*(u++i)), access(u*i)))
                           and ordered(u, u++i, j), c),
w = merge(u, v) implies interleaving(u, v, w) and
      (ordered(u) and ordered(v) implies ordered(w)),
w = merge(u, v, c) implies interleaving(u, v, w) and
      (ordered(u, c) and ordered(v, c) implies ordered(w, c)),
count(u, e) = count(u, begin(u), end(u), e),
valid(range(u, i, j)) implies
  count(u, i, j) = if i = j then 0
                   else if access(u*i) = e then 1 + count(u, u++i, j)
                        else count(u, u++i, j),
permutation(u, v) = ((for e: elements) count(u, e) = count(v, e)),
w = sort(u) implies
   permutation(w, u) and ordered(w) and
   ((for i: iterators) i in range(w, begin(w), end(w)) implies w*i = u*i).
w = sort(u, c) implies
   permutation(w, u) and ordered(w, c) and
   ((for i: iterators) i in range(w, begin(w), end(w)) implies w*i = u*i).
```

◇

Used in part 1.

16 Associative Container Concepts

⟨Associative container concept 20a⟩ ≡

```
Definition: Associative-container
refines Basic-sequence [with Bidirectional-iterator as Forward-iterator];
introduces count(containers, elements) -> naturals,
  count(containers, iterators, iterators, elements) -> naturals (private),
  find(containers, elements) -> iterators,
  erase(nonempty-containers, elements) -> containers,
  equal-range(containers, elements) -> (iterators, iterators),
  has-key(objects, elements) -> bool,
  equiv(elements, elements) -> bool;
requires (for i, j: iterators; c: containers; k: elements)
  count(c, k) = count(c, begin(c), end(c), k),
  valid(range(c, i, j)) implies
    count(c, i, j, k) = if i = j then 0 else
                        if c*i has-key k then 1 + count(c, c++i, j, k) else
                        count(c, c++i, j, k),
  (i, j) = equal-range(c, k) implies valid(range(c, i, j)) and
     if count(c, k) = 0 then
        i = end(c) and j = end(c)
     else
       ((for q: iterators) q in range(c, begin(c), end(c)) implies
           (q has-key k) = (q in range(c, i, j))),
  (i, j) = equal-range(c, k) and constant(c) implies
     points-to-constant(i) and points-to-constant(j),
  i = find(c, k) implies
      if count(c, k) = 0 then i = end(c)
      else i in range(c, begin(c), end(c)) and c*i has-key k
           and count(c, begin(c), i, k) = 0,
  constant(c) implies points-to-constant(find(c, k)),
  erase(c, k) = erase(c, i, j) where (i, j) = equal-range(c, k).
◇
```

Used in part 1.

16.1 Sorted Versus Hashed Associative Containers

⟨Sorted associative container concept 20b⟩ ≡

```
Definition: Sorted-associative-container
refines Associative-container;
uses Strict-weak-order [with key-comp as <];
introduces
  value-compare(objects, objects) -> bool,
  lower-bound(containers, elements) -> iterators,
  upper-bound(containers, elements) -> iterators;
requires (for c: containers; x, x1: objects; k, k1: elements)
  (x value-compare x1) = (k key-comp k1) where x has-key k and x1 has-key k1,
  ((for p, q: indices) p in range(c, least, least + size(c))
     and q in range(c, least, least + size(c)) and p < q
     implies not(key-comp(access(c, q), access(c, p)))),
  (k equiv k1) = (not(key-comp(k, k1)) and not(key-comp(k1, k))),
  lower-bound(c, k) = i where (i, j) = equal-range(c, i, j),
  upper-bound(c, k) = j where (i, j) = equal-range(c, i, j).
◇
```

Used in part 1.

⟨Hashed associative container concept 20c⟩ ≡

```
Definition: Hashed-associative-container
refines Associative-container;
```

```
introduces hash(elements) -> naturals;
requires (for c: containers; k, k1: elements)
  (k equiv k1) = (k = k1).
◇
```

Used in part 1.

16.2 Multiple Versus Unique Associative Containers

⟨Multiple associative container concept 21a⟩ ≡

```
Definition: Multiple-associative-container
refines Associative-container;
introduces
  insert(containers, elements) -> (containers, iterators);
requires (for c: containers; e: elements)
  (c1, i1, b) = insert(c, e) implies
     size(c1) = size(c) + 1 and access(c1*i1) = e
     and ((for p, q: indices)
            (p in range(least, position(c1, i1))
                implies access(c1, p) = access(c, p))
            and (q in range(position(c1, i1), least + size(c))
                implies access(c1, next(q)) = access(c, q))).
◇
```

Used in part 1.

⟨Unique associative container concept 21b⟩ ≡

```
Definition: Unique-associative-container
refines Associative-container;
introduces
  insert(containers, elements) -> (containers, iterators, bool);
requires (for c: containers; e: elements)
  count(c, e) = 0 or count(c, e) = 1,
  (c1, i1, b) = insert(c, e) implies
   b = (find(c, e) = end(c)) and
   if b then
     size(c1) = size(c) + 1 and access(c1*i1) = e
     and ((for p, q: indices)
            (p in range(least, position(c1, i1))
                implies access(c1, p) = access(c, p))
            and (q in range(position(c1, i1), least + size(c))
                implies access(c1, next(q)) = access(c, q)))
   else
     c1 = c and i1 = i.
◇
```

Used in part 1.

16.3 Simple Versus Pair Associative Containers

⟨Simple associative container concept 21c⟩ ≡

```
Definition: Simple-associative-container
refines Associative-container;
requires (for i: iterators; e, k: elements)
  points-to-constant(i),
  (e has-key k) = (e equiv k).
◇
```

Used in part 1.

⟨Pair and associative-pair concepts 21d⟩ ≡

```
Abbreviation: First is Assignable [with first-objects as objects].

Abbreviation: Second is Assignable [with second-objects as objects].

Definition: Pair
refines Assignable [with pairs as objects];
uses First, Second;
introduces construct(first-objects, second-objects) -> pairs,
  first(pairs) -> first-objects,
  second(pairs) -> second-objects;
requires (for f: first-objects; s: second-objects)
  first(construct(f, s)) = f,
  second(construct(f, s)) = s.

Abbreviation: Key is Object [with keys as objects].

Definition: Associative-pair
refines Pair [with Key as First, keys as first-objects];
requires (for p: pairs)
  constant(first(p)).
◇
```

Used in part 1.

⟨Pair associative container concept 22a⟩ ≡

```
Definition: Pair-associative-container
refines Associative-container [with Associative-pair as Accessible,
                                    pairs as objects];
requires (for x: objects; k: elements)
 (x has-key k) = (first(x) equiv k).
◇
```

Used in part 1.

16.4 Combinations of Associative Container Concepts

⟨Combinations of associative container concepts 22b⟩ ≡

```
Definition: Set-associative-container
refines Simple-associative-container, Unique-associative-container,
        Sorted-associative-container.

Definition: Multiset-associative-container
refines Simple-associative-container, Multiple-associative-container,
        Sorted-associative-container.

Definition: Map-associative-container
refines Pair-associative-container, Unique-associative-container,
        Sorted-associative-container;
introduces brackets(containers, elements) -> objects;
requires (m, m1: containers; i: iterators; k: elements)
  brackets(m, k) = second(m1*i)
       where (m1, i, b) = insert(m, construct(k, construct())).

Definition: Multimap-associative-container
refines Pair-associative-container, Multiple-associative-container,
        Sorted-associative-container.

Definition: Hashed-set-associative-container
refines Simple-associative-container, Unique-associative-container,
        Hashed-associative-container.
```

```
Definition: Hashed-multiset-associative-container
refines Simple-associative-container, Multiple-associative-container,
        Hashed-associative-container.

Definition: Hashed-map-associative-container
refines Pair-associative-container, Unique-associative-container,
        Hashed-associative-container.

Definition: Hashed-multimap-associative-container
refines Pair-associative-container, Multiple-associative-container,
        Hashed-associative-container.
◇
```

Used in part 1.

[[Need to add to the above concepts the definition of iterator validity, and to expand the discussion.]]

References

[1] Deepak Kapur and David R. Musser, *Tecton: A Framework for Specifying and Verifying Generic System Components*, RPI Computer Science Department Technical Report 92-20, Troy, NY, July 1992.

[2] Deepak Kapur and David R. Musser, *Examples of Tecton Concept Descriptions (Working Paper)*, June 1992.

[3] [[STL reference 1]]

[4] [[STL reference 2]]

[5] [[C++ standard]]

[6] Silicon Graphics Inc., *The Standard Template Library Programmer's Guide*, http://www.sgi.com/Technology/STL.