Charles: An STL for Ada95

Matthew J Heaney
mailto:matthewjheaney@earthlink.net

Charles is a container and algorithms library for Ada95, modeled closely on the C++ STL. The library provides both sequence and associative containers, and specifies the time and space semantics of each container. Charles is flexible and efficient, and its design has been guided by the philosophy that a library should stay out of the programmer's way.

The elements of a "sequence" container element are stored in linear order, at positions specified by the user. Each sequence container optimizes the cost of insertions differently. A *vector* is the most space-efficient container, and is specifically optimized for insertions at the back end. A *deque* is slightly less storage efficient, but is optimized for insertions at both the front and back ends. The vector and deque containers both provide random access of elements. A *list* is optimized for insertion at any position, but does not provide random access. (Charles does not have dedicated "stack" or "queue" containers, because you can achieve the same effect by simply appending or deleting items from the end of one of the more general sequence containers.)

The "associative" containers associate a key with each element, and then store elements in order by key. For a *set* container an element is its own key, but a *map* container allows you to index an item by some other arbitrary type (in this sense a map is a generalization of an array). Both hashed and sorted associative containers are provided. A hashed container has an average time complexity that is constant, and a sorted container guarantees logarithmic time complexity even in the worst case.

Containers are important because they provide a means of storing and retrieving elements, but ultimately it is the elements in which we are interested, not the container. An *iterator* is the means by which a container allows its elements to be queried, modified, or otherwise manipulated without exposing the representation of the container. The iterator mechanism in Charles is completely general, and an algorithm written in terms of an iterator works for *any* container with the requisite iterator operations.

The canonical container is the vector, which is simply a linear sequence of items. The generic package for vectors has separate generic formal types for the index (which must be discrete) and the element, so a vector is similar to an unbounded array. The vector container supports random access of its elements (meaning that index selection has unit time complexity), and is optimized for insertions at the back end of container. For example:

```
V : Integer_Vectors.Container_Type;
...
Append(V, New_Item => 42);  -- canonical insertion operation for vectors
```

A vector is implemented using an array, and thus has the best storage efficiency among all the containers. Insertion operations automatically expand the internal array when the vector has reached capacity. However, if you know the ultimate size of the vector prior to insertion, then you can use Resize to perform the allocation in advance, which is more efficient.

For a vector and deque, positions are specified using the discrete index type:

```
   Insert (V, Before => I, New_Item => X);
   X := Element (V, Index => I);
   Replace_Element (V, Index => J, By => Y);
```

For a vector, the insertion at the back end has (amortized) constant time complexity. However, as the insertion position approaches the front end, the time complexity becomes linear because all the elements from the insertion position to the back of the vector have to slide up to make room for the new item. (This is why there is no Prepend operation for vectors, although the effect of such an operation can achieved by using Insert to add the item before the position Index_Type'First).

In the example above, we used the selector function Element to return the value of the element at the indicated position. However, returning a copy of the element is not a sufficiently general mechanism, as efficiency issues might prohibit copying the element simply to query its value. For modifying an element, Replace_Element isn't adequate either since that operation only allows you to assign a completely new value to an element. To satisfy the need for direct, in-place manipulation of elements, all the containers in Charles have an additional generic selector function that accepts an access type as a generic formal type, and which returns an access value designating a variable view of the actual element:

```
declare
   function To_Access is new Generic_Element (Integer);
   E : Integer renames To_Access (V, Index => I).all;
begin
   E := E + 1;  -- modify element in-place
end;
```

A "deque" (short for **d**ouble-**e**nded **que**ue) is similar to a vector, with the difference that insertion of an element at the front of the container has only constant time complexity. It is implemented as an unbounded array of pointers to fixed-size blocks, with each page comprising a fixed number of elements. Prepend works by adding new elements to the first page, and decrementing an offset that keeps track of which element on the page is the first active element. When there is no more room on the front page, then a new page is allocated and the offset is reset.

The deque container also allows elements to be inserted (and deleted) at any position, but the time complexity becomes linear as the insertion position approaches the middle of the container. One difference from a vector is that to make room for the new element, the existing elements are moved toward the end of the deque that is closest to the insertion position.

Deques are especially useful when you have a large number of items to insert, but the total number of items cannot be determined prior to insertion. A vector would need to allocate a new, longer array each time it reaches capacity, which requires copying the existing elements onto the new array. But when a deque reaches capacity (really, its last page becomes full), it simply allocates a new page, which does not affect existing items and hence is much more efficient. A deque also doesn't need large contiguous regions of virtual memory as a vector does.

Lists are optimized for insertions at any position. All insertions and deletions, in the middle or at either end, have constant time complexity. However, random access is not supported, and navigating among list positions has a time complexity proportional to the distance between positions. For referring to positions in the list, we use an iterator:

```
   I : constant Iterator_Type :=  Last (List);
   E : Element_Type renames To_Access (I).all;
```

To insert an item at a certain position, we specify the position using an iterator value:

```
   Insert (List, Before => Iterator, New_Item => Item);
```

The Find operation returns an iterator value, which we can test against the distinguished iterator value Back (which designates the logical element immediately beyond the last position) to determine whether the search was successful:

```
   I : constant Iterator_Type := Find (List, Item);
begin
   if I /= Back (List) then ... --search succeeded
```

If you need to move an element within a list, or from one list onto another list, the Splice operation is optimal, because it manipulates the internal node of storage containing the element, rather than the element itself:

```
   Target, Source : T_Lists.Container_Type;
   I, J : T_Lists.Iterator_Type;
   ...
   Splice (Target, I, Source, J);
```

Here, Splice moves (*not* copies) the item designated by iterator J in list container Source, onto the list Target just prior to the element designated by iterator I. This preserves the identity of element that was moved. Additional overloadings of Splice generalize this behavior to move an entire range of elements.

The list container also provides operations for reversing the list, sorting the elements, removing duplicates, filtering, and for merging two (sorted) lists. The sort operation is stable (meaning that the relative order among equal items is preserved across a sort).

The library has both singly-linked and doubly-linked list containers. Both forward and reverse iteration over a double list is provided, but only forward iteration is possible for single lists. The singly-linked list is more space efficient, however, because the storage overhead per node is smaller. The single list also caches a pointer to the last node, so that appending an item is a constant time operation, thus making the single list attractive for use as a traditional "queue" data structure.

Note that the list containers are "monolithic" data structures, the same as for all the containers in the library. There is no structure sharing, and list assignment is by value, not by reference. The

name *list* simply emphasizes the fact that this container is optimized for constant-time insertion and deletion of elements at arbitrary positions. The semantics of this abstraction are *not* the same as for the similarly-named "polylithic" data structure in functional languages as LISP.

In addition to the sequence containers, the Charles library also has sets and maps that associate a key with each element. For a set the element is its own key, but for a map there is a separate key type. For the *sets* and *maps* keys must be unique, while for the *multisets* and *multimaps* keys can be the same.

There is only a subtle difference in how sets and maps are implemented. For a set, the element is its own key, and the set doesn't allocate space for anything other than the element. For a map, the key is separate piece of data, and the map is implemented as key/element pairs. For a sorted container, the elements are stored in a balanced red-black tree, and for a hashed container, the elements are stored in a hash table that expands as necessary to preserve a constant load factor.

In a sorted container, keys are ordered according to a less-than relation passed as a generic formal operation. A hashed container uses a hash function to scatter keys throughout the table. When the capacity is reached, the container automatically expands the hash table in order to maintain the load factor. Hash table lengths always correspond to a prime number, as this produces better scatter.

Insertion of an element into a set is done in the normal way:

```
Set : Integer_Sets.Container_Type;
...
Insert (Set, New_Item => 42);
```

For a map, you must also specify the key associated with that element:

```
Map : String_Integer_Maps.Container_Type;
...
Insert (Map, Key => "Adams", New_Item => 42);
```

The insertion operation is overloaded to return an iterator that designates the element just inserted:

```
procedure Op (Set : Integer_Sets.Container_Type) is
   I : Integer_Sets.Iterator_Type;
   B : Boolean;
begin
   Insert (Set, New_Item => E, Iterator => I, Success => B);
   ...
end;
```

This example illustrates that insertion into a set or map is conditional, since those containers require that keys be unique. Insert returns an indication of whether the key was already in the container. If the indicator is True, then the key was not in the container, and the element (or key/element pair) was successfully inserted into the container. If the indicator is False, then the key was already present, and the iterator designates the existing key. Note that in a multiset or

multimap, insertion is not conditional since those containers allow keys to be the same, and therefore there is no need for a success indicator (because insertion is always succeeds).

In order to store a key in a map, it must have a definite subtype. One issue is that some keys are more naturally represented using indefinite subtypes, e.g. a name having type String. To use the map the client would first have to convert the key to the definite subtype used to instantiate the package, and then call the map operation. However, this is both inefficient and syntactically heavy. Since maps with string keys are nearly ubiquitous, the library provides special map containers that have type String as the key type. The packages also accept a string comparison operation as a generic formal, which would (for example) allow key comparisons to be case-insensitive.

The idiom for finding a key in an associative container is the same as for the sequence containers:

```
procedure Op (Map : String_Integer_Maps.Container_Type) is
   I : String_Integer_Maps.Iterator_Type := Find (Map, Key);
begin
   if I /= Back (Map) then -- key was found
      E := Element (I);
   ...
end Op;
```

For the multiset or multimap, Find returns an iterator designating the first member of the group of equal keys, which are always (logically) adjacent.

To visit the members of a group of equal keys in a hashed multimap (or multiset), there's a generic operation designed specially for this purpose:

```
procedure Op (M : Multimap_Subtype) is
   procedure Process (I : Iterator_Type) is
   begin
      Do_Something (Element (I));
   end;

   procedure Iterate is new Generic_Equal_Range (Process);
begin
   Iterate (M, Key => K);  -- visit all the elements having key K
end;
```

The sorted sets and maps have an operation called Lower_Bound, which returns the smallest key in the container equal or greater than a specified key, and another operation called Upper_Bound, to return the smallest key greater than a specified key. These operations are useful for finding the key in the container that is either equal to a specified key (if the key is in the container) or would be adjacent to it (if the key is not in the container). This behavior is like a search operation that returns a partial match. If you had a set of test scores, for example, then you could use these operations to, say, iterate over the scores within each tenth percentile.

These are also the operations one would use to walk a group of equivalent keys in a sorted multimap (or multiset):

```
procedure Op (M : Multimap_Subtype) is
   I : Iterator_Type := Lower_Bound (M, Key => K); -- first of group
   J : Iterator_Type := Upper_Bound (M, Key => K); -- one beyond last
begin
   while I /= J loop
      Process (Element (I));
      Increment (I);
   end loop;
end Op;
```

The sorted containers are implemented using a balanced tree and therefore the time complexity of insertions is O(log n). However, if you have a key and know its nearest neighbor in the container, there is a special version of Insert that accepts this information as a hint about where to begin searching for the key, and if the hint is successful the time complexity is only O(1). This is useful for copying a sequence of sorted items into a sorted set (say), since you can use the result of the current insertion as the hint for the next insertion.

The set also has a nested generic package called Generic_Keys that allows you to get a key-view of the element. This is useful when the element is record, and the order relation for elements is computed from one of the components of the record. The canonical example is an employee set:

```
type Employee_Type is record
   SSN : SSN_Type;  -- this is the key
   ...
end record;

function "<" (L, R : Employee_Type) return Boolean is
begin
   return L.SSN < R.SSN;
end;

package Employee_Sets is
   new Charles.Sets.Sorted.Unbounded (Employee_Type, "<");
```

We can now add an employee to the set in the normal way:

```
Employees : Employee_Sets.Container_Type;
...
procedure Hire is
   E : Employee_Type;
begin
   E.SSN := ...;
   E.Name := ...;
   E.Home_Phone := ...;
   ...
   Insert (Employees, New_Item => E);
end;
```

Now suppose we want to look up an employee by his social security number. We can't use Find to do this, since Find accepts type Employee_Type, not SSN_Type. However, SSN_Type is

what we used to compute the order relation for Employee_Type, so we can use it as the generic actual type to instantiate Generic_Keys:

```
function "<" (L : SSN_Type; R : Employee_Type) return Boolean is
begin
   return L < R.SSN;
end;
...
package SSN_Keys is new Employee_Sets.Generic_Keys (SSN_Type, "<", ">");
```

Now we can use the key-oriented operations to manipulate the set:

```
procedure Op (SSN : SSN_Type) is
   I : Iterator_Type := SSN_Keys.Find (Employees, Key => SSN);
begin
   if I /= Back (Employees) then ...;
end;
```

This is another example of how a sets and maps are very similar. Really they only differ with respect to where the key is stored.

A reusable abstraction should be as flexible, efficient, and safe as possible, but these goals are often in conflict. This tension is particularly acute in the design of iterators, since they must provide access to elements of the container without exposing its representation. In Charles, iterators have been designed to be both flexible and efficient. However, they confer no greater safety benefits beyond what is provided by built-in access types. Where greater type safety is desired (say, to detect dangling references), then the client can implement that higher-level abstraction himself using the lower-level primitives provided by the library.

To see how container iterators are designed, we can compare them to how access types are used to traverse a simple linked list:

```
declare
   Node : Node_Access := List.Head;
begin
   while Node /= null loop
      Process (Node.Element);
      Node := Node.Next;
   end loop;
end;
```

Substituting iterator primitives for access types, we can rewrite the example like this:

```
declare
   I : Iterator_Type := First (List);
   J : constant Iterator_Type := Back (List);
begin
   while I /= J loop
      Process (Element (I)); --or Process (To_Access (I).all)
      I := Succ (I);
   end loop;
end;
```

This schema can be generalized to work for any container besides a list, and this is in fact how iterators are implemented in Charles. The fact that the iterator is non-limited and definite means you can implement any other type in terms of the iterator. This is important because real systems are built from the bottom up, by assembling complex abstractions from simpler primitives.

This generality allows us to write generic algorithms that work for any container, even built-in arrays. For example:

```
generic
   type Iterator_Type is private;
   with function Succ (Iterator : Iterator_Type) return Iterator_Type is <>;
   with function Pred (Iterator : Iterator_Type) return Iterator_Type is <>;
   with procedure Swap (L, R : Iterator_Type) is <>;
procedure Charles.Algorithms.Generic_Reverse (First, Back : Iterator_Type);
```

Notice that the container is *not* imported as a generic formal, since it is abstracted-away behind the iterator interface. We can instantiate it for use with an array as follows:

```
procedure Reverse_Array (A : in out Array_Type) is
   procedure Swap (I, J : Positive) is
      E : constant ET := A (I);
   begin
      A (I) := A (J);
      A (J) := E;
   end;

   procedure Reverse_Array is
      new Generic_Reverse
        (Iterator_Type => Positive,
         Succ          => Integer'Succ,
         Pred          => Integer'Pred,
         Swap          => Swap);
begin
   Reverse_Array
    (First => A'First,
     Back  => A'First + A'Length);
end Reverse_Array;
```

We could just have easily used this same algorithm to reverse the elements of a vector or a list or even a map. Charles has an entire suite of generic algorithms similar to the example above.

The Charles library satisfies the need for a general-purpose library having the right balance between flexibility and ease of use. It allows developers to program at a level of abstraction higher than arrays or simple linked lists, using efficient, composable primitives that have specified behavior. More recently Charles has also served as the basis of the design of the standard container library that will be included in the next revision of the Ada language.

The source code for Charles may be accessed via the CVS repository http://charles.tigris.org . *All the sources use the GNAT Modified GPL licensing scheme.*