Charles: A Data Structure Library for Ada95

Matthew Heaney

Ada-Europe 2003

Toulouse, France

Library Design

- The library designer has many goals: maximizing flexibility, generality, efficiency, safety, ease of use, simplicity, elegance, etc.
- The library designer must arbitrate among these goals, which are often in conflict.

Flexibility

- The library designer can’t anticipate every library user’s specific need, so it’s best to provide flexible primitives that can be easily combined.
- Flexibility is often in conflict with ease of use and safety, so the designer must sometimes walk a fine line.

Efficiency

- The library must be as least as efficient as what a user can write himself, otherwise he won’t use the library.

Safety

- To make a completely unbreakable abstraction, you’ll have to give something else up, such as flexibility or efficiency.
- The library designer should defer to the library user how best to provide safety.
- Flexible and efficient library primitives can be combined to make a safe abstraction, but the opposite is not true.

Design Philosophy

- A library should stay out of the user's way.
- It's *easy* to do common things, and *possible* to less common things.
- Library primitives are easily composable.

Containers

- Sequence containers (vectors, queues, lists) store unordered elements, which are inserted at specified positions.
- Associative containers (sets, maps) store elements in key order.

Time Complexity

- The time complexity of operations is specified. It is not a implementation detail.
- Different containers have different time and space semantics. You instantiate the component that has the properties you desire.

Static Polymorphism

- It is helpful to make a distinction between instantiating a generic component versus using the instantiated component.
- The generic formal region of components can differ. However, once the component has been instantiated, then the differences more or less disappear, because each component has more or less the same interface.

Sorted Set

```
generic

   type Element_Type is private;

   with function "<" (L, R : Element_Type)
      return Boolean is <>;

   with function "=" (L, R : Element_Type)
      return Boolean is <>;

package Charles.Sets.Sorted.Unbounded is

   type Container_Type is private;
   type Iterator_Type is private;
```

Hashed Set

```
generic

   type Element_Type is private;

   with function Hash (Item : Element_Type)
      return Integer'Base is <>;

   with function "=" (L, R : Element_Type)
      return Boolean is <>;

package Charles.Sets.Hashed.Unbounded is

   type Container_Type is private;
   type Iterator_Type is private;
```

Sorted Set or Hashed Set?

```
procedure Op
  (Set : in out Element_Sets.Container_Type) is

   I : Element_Sets.Iterator_Type;
begin
   Insert (Set, New_Item => E);

   I := Find (Set, Item => E);

   Delete (Set, Item => E);
end;
```

Iterators

- Elements are everything. Containers are nothing.
- The purpose of an iterator is to provide access to the elements in a container, without exposing container representation.
- Elements are not a hidden detail, and the library takes pains to ensure that access to elements is easy and efficient.

Machine Model

- Iterators allow the container to be viewed as an abstract machine, containing elements that are logically contiguous.
- You navigate among element "addresses" using an iterator, and "dereference" the iterator to get the element at that address.

Iterator Type

- For full generality, the iterator type is definite and nonlimited. It thus has the same properties as a plain access type.
- An indefinite or limited iterator type is not sufficiently general, among other reasons because we wouldn't be able to store an iterator object as a container element.

Iterator Representation

- An iterator type hides container representation details.
- An iterator is implemented as a thin wrapper around an access type that designates a node of internal storage.
- An iterator type does not confer any safety benefits above and beyond what is available for an ordinary access type.

Half-Open Range

- An iterator pair is used to denote a half-open range of (logically) contiguous elements.
- The first iterator denotes the first element in the range, and the second iterator denotes the (logical) element one beyond the last element of the range.
- Falling off the end onto the sentinel indicates completion of the iteration.

Active vs. Passive Iteration

- During active iteration, advancement of the iterator value is controlled by the client.
- During passive iteration, iterator advancement is controlled by the operation.
- An active iterator is appropriate when more than one container is being visited simultaneously, although the approaches can be combined.

```
procedure Op (C : Container_Subtype) is
   I : Iterator_Type := First (C);
   J : constant Iterator_Type := Back (C);
begin
   while I /= J loop
      declare
         E : constant Element_Type :=
           Element (I);
      begin
         Do_Something (E);
      end;

      I := Succ (I);
   end loop;
end Op;
```

```
procedure Op (C : Container_Subtype) is
   I : Iterator_Type := Last (C);
   J : constant Iterator_Type := Front (C);
begin
   while I /= J loop
      declare
         E : Element_Type renames
           To_Access (I).all;
      begin
         Do_Something (E);
      end;

      I := Pred (I);
   end loop;
end Op;
```

```
procedure Op (C : Container_Subtype) is

   procedure Iterate is
     new Generic_Select_Elements
       (Process => Do_Something);
begin
   Iterate (First (C), Back (C));
end Op;
```

```
procedure Op (C : Container_Subtype) is

   procedure Iterate is
     new Generic_Reverse_Select_Elements
       (Process => Do_Something);
begin
   Iterate (First (C), Back (C));
end Op;
```

Vectors

- Provides random access to elements.
- Complexity of insertion at back end is amortized constant time.
- Internal array expands as necessary to store more elements. New size of array is a function of its current size.

```
declare
   V : Vector_Subtype;
begin
   Push_Back (V, New_Item => E);
   Pop_Back (V);

   Insert (V, Before => I, New_Item => E);
   Delete (V, Index => I);

   E := Element (V, Index => I);
   Replace_Element (V, Index => I, By => E);
end;
```

Deques (**D**ouble-**E**nded **Que**ue)

- Like a vector, it provides random access to elements.
- Unlike a vector, insertion at front end has constant time complexity.
- Insertion slides elements towards the nearest end to make room for the new item.

```ada
declare
   D : Deque_Subtype;
begin
   Push_Back (D, New_Item => E);
   Pop_Back (D);

   Push_Front (D, New_Item => E);
   Pop_Front (D);

   E := Element (D, Index => I);
   Replace_Element (D, Index => I, By => E);
end;
```

Lists

- Insertion has constant time complexity, at all positions.
- No random access.
- The list container is monolithic, *not* polylithic (a la LISP); there is no structure sharing.

```
declare
   L : List_Subtype;
   I : Iterator_Type;
begin
   Insert
     (Container => L,
      Before    => Back (L),
      New_Item  => E,
      Iterator  => I);

   E := Element (I);
   Replace_Element (Iterator => I, By => E);

   Delete (L, Iterator => I);
end;
```

Variable View of Elements

- The Element function returns a copy of the element in the container.
- The Replace_Element procedure assigns a new value to the element in the container.
- This is not sufficient: we often need a way to modify the element, not simply replace its value. Example: a container whose elements are another container.

Dereferencing an Iterator

- The Generic_Element function returns an access object that designates the actual element, allowing in-place modification.
- Has the sense of a dereference operator.
- This is the best we can do in the absence of reference types a la C++.

```
type Element_Access is
   access all Element_Subtype;

function To_Access is
   new Generic_Element (Element_Access);

procedure Op (I : Iterator_Type) is
   L : List_Subtype renames
     To_Access (I).all;
begin
   Push_Back (L, E);
end;
```

Associative Containers

- Associative containers (sets, maps) store elements ordered by key.
- There are both sorted (tree-based) and hashed versions.
- Multimaps allow *keys* to be equivalent (sorted) or equal (hashed). Multisets allow *elements* to be equivalent or equal.

Worst Case vs. Average Case

- Sorted associative containers guarantee that insertion has worst-case logarithmic time complexity.
- Hashed associative containers have unit time complexity on average.

Strict Weak Ordering

- During insertion, keys in a sorted set or map are compared for “equivalence,” not equality.
- Keys are the “same” if the following relation is true:

$$\text{not } (L < R) \text{ and not } (R < L)$$

Sets vs. Maps

- There is only a subtle difference between a set and a map: a set has only an element, and a map has a key/element pair.

Maps

- Elements are stored in key order.
- Internally, keys and elements are stored as pairs.
- Appropriate for elements whose key is separate from element.

Membership Tests

- The Find operation is used to determine whether an element is in the map.
- Find returns an iterator as its result. If the iterator has the distinguished value Back, then the search failed and the element is not in the map.
- Otherwise, the iterator designates the key/element pair whose key matched.

```
procedure Op (M : in out Map_Subtype) is
   I : Iterator_Type;
begin
   I := Find (M, Key => K);

   if I /= Back (M) then
      declare
         E : Element_Subtype renames
            To_Access (I).all;
      begin
         ... -- modify E as desired
      end;
   end if;
end Op;
```

Unconditional Insertion

- Insert searches the map to determine whether the key is a member.
- If the key already in the map, then the element associated with that key is replaced by the new value.
- Otherwise, the new key/element pair is inserted in the map.

```
procedure Op (M : in out Map_Subtype) is
   I : Iterator_Type;
begin
   Insert
     (Container => M,
      Key       => K,
      New_Item  => E,
      Iterator  => I);
   ...
   Delete (M, Iterator => I);
   --or: Delete (M, Key => K);
end;
```

Conditional Insertion

- Suppose we don’t want to modify the element if its key is already in the map?
- We could try to Find the key, and if it's not found then Insert the key/element pair in the map. However, that would be inefficient, because Insert must perform a search internally, thus duplicating the search performed by Find.
- Better to simply use Insert, and let it report whether the insertion was successful.

```
procedure Op (M : in out Map_Subtype) is
   I : Iterator_Type;
   B : Boolean;
begin
   Insert
     (Container => M,
      Key       => K,
      New_Item  => E,
      Iterator  => I,
      Success   => B);

   if B then ...;
end Op;
```

Conditional Insertion (cont'd)

- If Insert returns success, then the key/element pair was inserted into the map, and the iterator designates the newly-inserted key/element pair.
- If Insert returns not success, then the key was already in the map, and the iterator designates the existing key/element pair, which is *not* modified.

Deletion

- An element can be deleted either by specifying its key, or by specifying an iterator that designates the element.

```
procedure Op (Map : in out Map_Subtype) is
   I : Iterator_Type;
begin
   Insert (Map, Key, E);
   Delete (Map, Key);

   Insert (Map, Key, E, Iterator => I);
   Delete (Map, Iterator => I);
end;
```

Multimaps

- A multimap allows multiple keys to be equivalent (sorted) or equal (hashed).
- There is no conditional insert, because all insertions succeed.
- Equivalent keys are contiguous. Equal_Range can be used to return an iterator pair that designates the range.

```
procedure Op (M : in Map_Subtype) is
   I, J : Iterator_Type;
begin
   Equal_Range
     (Container => M,
      Key       => K,
      First     => I,
      Back      => J);

   while I /= J loop
      declare
         E : Element_Subtype renames
            To_Access (I).all;
      begin ...;

      I := Succ (I);
   end loop;
```

Sets

- Like a map, except that an element is its own key.
- There is no separate key object, and only the element is stored in the container.
- Find can be used to test an element for membership in a set. Generic_Find can be used to test an element's key directly.

```
type Employee_Type is record
   SSN : SSN_Type; ... end record;

procedure "<" (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, "<");

procedure Op (S : in Employee_Set_Subtype) is
   I : Iterator_Type := Find (S, Item => E);
```

```
function "<"
   (E : Employee_Type; SSN : SSN_Type)
    return Boolean is
begin
   return E.SSN < SSN;
end;

function Find is
   new Generic_Find (SSN_Type);

procedure Op (S : in Employee_Set_Subtype) is
   I : Iterator_Type := Find (S, Key => SSN);
begin
   if I /= Back (S) then …;
```

Set Manipulation Via Key

- Searching by key is useful when the set element is an iterator, that designates the real element, which is stored in some other container (as a list, say, or even another set).
- This allows the same elements to be virtual members (via an iterator proxy) of different sets simultaneously, each of which orders the elements differently.

Multiset

- Like a set, except that multiple elements are allowed to be equivalent (sorted) or equal (hashed).
- As for a multimap, there is no conditional insertion because all insertions succeed.
- Equal_Range is used to return an iterator pair that designates the contiguous range of elements equivalent to an item.

```
procedure Op (S : in out Set_Subtype) is
   I, J : Iterator_Type;
begin
   Insert (S, New_Item => E);

   Equal_Range
     (Container => S,
      Item      => E,
      First     => I,
      Back      => J);

   while I /= J loop ...;

   Delete (S, Item => E);
```

What Is A Set?

- Charles (and the STL) takes a very liberal attitude about what is a “set” container.
- Basically, any container whose elements are sorted qualifies as a set.
- This means that we can apply generic set algorithms to any container having the requisite set properties.

```
procedure Union
  (Set  : in     Element_Sets.Container_Type;
   List : in     Element_Lists.Container_Type;
   EA   :    out Element_Array;
   Last :    out Natural) is

   procedure Process (I : Element_Sets.Iterator_Type) is
   begin
      Last := Last + 1;
      EA (Last) := Element (I);
   end;

   function Is_Less
     (SI : Element_Sets.Iterator_Type;
      LI : Element_Lists.Iterator_Type) return Boolean is
   begin
      return Element (SI) < Element (LI);
   end;
   ...
   procedure Union is
      new Charles.Algorithms.Generic_Set_Union_2
        (Element_Sets.Iterator_Type,
         Element_Lists.Iterator_Type);
begin
   Last := Integer'Pred (EA'First);
   Union (First (Set), Back (Set), First (List), Back (List));
end;
```