THE UNIVERSITY *of York*

Ada 2005 Standard Container Library

Matthew J Heaney
On2 Technologies, Inc
email: *matthewjheaney@earthlink.net*

24/6/2005

Reliable Software Technologies, Ada Europe 2005. York, UK. 20-24 June 2005
http://www.ada-europe.org/conference2005.html

Useful Links

http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-20302.TXT/

http://charles.tigris.org/source/browse/charles/src/ai302/

Container Taxonomy

- Sequence containers (vectors, lists) store elements at specified positions.
- Associative containers (sets, maps) store elements in key order.
- There are alternate forms of all containers, for storing indefinite elements and keys.
- The associative containers have both hashed and ordered forms.

Time Complexity

- The standard specifies the time complexity of operations. It is *not* an implementation detail. Indeed, this is practically *the* reason why the library comprises a suite of containers.
- Different containers have different time semantics. You choose whichever container has the particular properties best suited to the needs of your particular application.

Static Polymorphism

- There is a distinction between instantiating a generic component, and using the instantiated component.
- The generic formal region of components can differ, but as far as using them goes, once instantiated then components are basically the same, since they have a more or less identical static interface. (They differ in their execution behavior, of course.)
- Yes, container types are tagged, but that's mostly to give you distinguished-receiver syntax. (Subprogram parameters that are tagged are also implicitly aliased.)

Ordered 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 Ada.Containers.Ordered_Sets is

   type Set is tagged private;
   type Cursor is private;
```

Hashed Set

```
generic

   type Element_Type is private;

   with function Hash (Element : Element_Type)
      return Hash_Type;

   with function Equivalent_Elements
     (L, R : Element_Type) return Boolean;

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

package Ada.Containers.Hashed_Sets is

   type Set is tagged private;
   type Cursor is private;
```

Ordered Set or Hashed Set?

```
procedure Op (S : in out Set) is
   C : Cursor;
begin
   S.Insert (New_Item => E);

   C := S.Find (Item => E);

   S.Delete (Item => E);
end;
```

Cursors and (Passive) Iterators

- Containers are nothing. Elements are everything.
- A container exists for no other purpose than to store and retrieve elements. Elements are *not* a hidden detail.
- A cursor (and passive iterator) provides access to the elements in a container, without exposing container representation.

Machine Model

- Cursors allow the container to be viewed as an abstract machine, with elements that are logically contiguous.
- You navigate among element "addresses" using a cursor, and "dereference" the cursor to get the element at that address.
- The cursor design effectively abstracts-away the container, since all you have are cursors, and elements designated by cursors.

Active Iterator (Cursor)

```
procedure Op (Container : in Container_Type) is
   C : Cursor := Container.First; -- factory
   E : Element_Type;
begin
   while Has_Element (C) loop
      E := Element (C);
      exit when Predicate (E);
      Do_Something (E);
      Next (C);  --or: C := Next (C);
   end loop;
end Op;
```

Active Iteration (Cursors)

- During active iteration, navigation among cursor positions is controlled by the client.
- An active iterator (cursor) is appropriate when more than one container (or more than a single element within the same container) is being visited simultaneously.
- Use an active iterator to terminate the iteration without visiting every element in the container.

Passive Iterator

```
procedure Op (Container : Container_Type) is

  procedure Process (C : Cursor) is
     E : constant Element_Type := Element (C);
  begin
     Do_Something (E);
  end;
begin
   Container.Iterate (Process'Access);
end Op;
```

Passive Iteration

- During passive iteration, advancement is controlled by the operation itself.
- The passive iterator visits every element in the container. (It's designed for the common case.)
- Potentially more efficient than an active iterator, since the passive iterator knows that it is visiting all elements in sequence, and hence can visit elements in a way that takes advantage of the container’s representation.

Reverse Iteration

- All containers (except hashed) allow iteration in both forward and reverse directions.

Active Iteration In Reverse

```
procedure Op (Container : in Container_Type) is
   C : Cursor := Container.Last; -- factory
   E : Element_Type;
begin
   while Has_Element (C) loop
      E := Element (C);
      exit when Predicate (E);
      Do_Something (E);
      Previous (C); --or: C := Previous (C);
   end loop;
end Op;
```

Passive Iteration In Reverse

```
procedure Op (Container : Container_Type) is

  procedure Process (C : Cursor) is
     E : constant Element_Type := Element (C);
  begin
     Do_Something (E);
  end;
begin
   Container.Reverse_Iterate (Process'Access);
end Op;
```

Constant View of Elements

- The Element function returns a copy of the element in the container.
- However, if what you really want to do is just query the element directly, then that function can be relatively inefficient if the element is expensive to copy (it's large, or controlled, etc).
- The operation Query_Element returns a constant view of the actual container element.

```
procedure Op (C : Cursor) is
   procedure Process (E : in Element_Type) is
   begin
      ... -- do something with E
   end;
begin
   Query_Element (C, Process'Access);
end;
```

Variable View of Elements

- The Replace_Element procedure assigns a new value to the element in the container.
- This operation alone is not general enough: we often need a way to modify the element in place, not simply replace its value.
- The operation Update_Element returns a variable view of the actual container element.

```
procedure Op (C : Cursor) is
   procedure Process (E : in out Element_Type) is
   begin
      ... -- do something with E
   end;
begin
   Update_Element (C, Process'Access);
end;
```

“Tampers with Elements”

- When Process.all is executing from within Query_Element or Update_Element, there are things you can’t do to the container: Clear, Move, Insert, etc. Basically anything that can change container cardinality is *verboten*.
- You also cannot call any operation that would replace the element you’re currently visiting: Swap, Replace_Element, etc.

Sequence Containers

- Insert, Append, Prepend
- Find, Reverse_Find, Contains
- Delete, Delete_First, Delete_Last
- Element, Query_Element
- Replace_Element, Update_Element
- First, First_Element, Last, Last_Element
- Swap, Move, Generic_Sorting
- Iterate, Reverse_Iterate

Vectors

- Provides random access to elements.
- Complexity of Append is amortized constant time. (But Prepend is linear time.)
- The vector model is that an internal array automatically expands as necessary to store more elements. (That’s the *model*. A vector doesn’t have to be implemented as an array.)

Index-based Operations

```
declare
   V : Vector;
   I : Extended_Index;
begin
   V.Append (New_Item => E);
   I := V.Last_Index;

   E2 := V.Element (Index => I);
   V.Replace_Element (Index => I, By => E3);

   V.Insert (Before => I, New_Item => E4);
   V.Delete (Index => I);

   V.Delete_Last;
end;
```

Cursor-based Operations

```
declare
   V : Vector;
   C : Cursor;
begin
   V.Append (New_Item => E);
   C := V.Last;

   E2 := Element (Position => C);
   Replace_Element (Position => C, By => E3);

   V.Insert (Before => C, New_Item => E4);
   V.Delete (Position => C);

   V.Delete_Last;
end;
```

Index- vs. Cursor-based Operations

- Vectors are unique in that they have both index-based and cursor-based operations.
- The cursor-based operations make it easier to switch between a vector and some other container (usually a list), and provide a uniform syntax for iteration (that applies to all containers). [See list ex. on p. 42.]

Capacity vs. Length

- The function Capacity returns the total amount of internal storage. A vector automatically increases the capacity during insertion, when the current length (number of elements) equals the current capacity.
- Reserve_Capacity tells the vector to preallocate a specified amount of internal storage. **If you know the total number of elements in advance of insertion, it's more efficient to reserve the necessary capacity, since the expansion of the internal array is done only once.**

```
procedure Copy (A : Array_Type) is
   V : Vector;
begin
   V.Reserve_Capacity (Capacity => A'Length);

   for I in A'Range loop
      V.Append (New_Item => A (I));
   end loop;
      ...
end Copy;
```

Set_Length

- You can also set the vector length explicitly. This can be used to either truncate the vector (and hence throw elements away), or expand the vector (in which case “empty” elements are appended).

```
procedure Copy (A : Array_Type) is
   V : Vector;  --or := To_Vector (A'Length);
   J : Extended_Index := No_Index;
begin
   V.Set_Length (Length => A'Length);

   for I in A'Range loop
      J := J + 1;
      V.Replace_Element(Index => J, By => A (I));
   end loop;
      ...
end Copy;
```

Insert_Space

- Insert_Space is another way to reserve capacity, by making room (“space”) in the middle of the vector, without having an actual element to insert.

```
procedure Copy
  (A : in     Array_Type;
   V : in out Vector;
   I : in     Extended_Index) is

   J : Extended_Index := I - 1;
begin
   V.Insert_Space (Before => I, Count => A'Length);
   -- dig the hole (no elements yet)

   for K in A'Range loop
      J := J + 1;

      V.Replace_Element (Index => J, By => A (K));
      -- fill the hole (with elements)
   end loop;
   ...
end Copy;
```

Inserting Multiple Elements

- All insertion and deletion operations have a count parameter (that defaults to 1) to control how many elements are inserted or deleted. There are additional overloadings of insertion operations, that accept a vector as the New_Item parameter.
- **In general, when you insert multiple elements into a vector, you should try to do it in a way that avoids repeated expansion of the internal array.** If you know how many elements you intend to insert, then either Reserve_Capacity first, or Insert_Space, or Set_Length, or specify the Count parameter of Insert.

Index-based Active Iteration

- As for any other container, you can use either a passive iterator or a cursor-based active iterator. However, since a vector also supports index-based operations, you can also use a for loop to iterate in the traditional way.

```
procedure Op (V : in Vector) is

   procedure Process
    (E : in Element_Subtype) is
   begin
     ... -- do something with E
   end Process;

begin -- Op

   for I in V.First_Index .. V.Last_Index loop
      V.Query_Element (I, Process'Access);
   end loop;

end Op;
```

Move

- The Move operation *moves*, not copies, the elements from one container to another.
- In the case of a vector, Move works by simply transferring the internal array. For other containers, Move is implemented similarly.
- Move makes it possible to use a container to assemble elements in one part of a system, and then move them to another part of the system.

```
L : Lists_Of_Vectors.List;
...
declare
   Src : Vector;
begin
   Src.Append (New_Item => E);
   ... --populate Src some more as appropriate

   L.Append (New_Item => Empty_Vector);

   declare
      procedure Process (Tgt : in out Vector) is
      begin
         Move (Source => Src, Target => Tgt);
      end;
   begin
      Update_Element (Last (L), Process’Access);
   end;
end;
```

Swap

- Swap (logically) exchanges a pair of elements.
- Mostly intended to take advantage of the representation of indefinite vectors, which allocate each element.
- Swap is implemented (in the indefinite vector case) by exchanging the internal pointers, which can be potentially more efficient than exchanging elements directly.

Find, Reverse_Find

- Every container has a Find operation to search for an element. For vectors and lists, Find performs a linear search from First to Last. (For the maps and sets, the search works differently, and is definitely *not* linear.)
- For the sequence containers, there's also a Reverse_Find, to search from Last to First.
- Vectors also have indexed-based versions: Find_Index and Reverse_Find_Index.
- Search operations for the sequence containers have a parameter (with a suitable default) to specify from where to begin the search.

Lists

- Insertion and deletion have constant time complexity at all positions.
- No random access.
- The list container is monolithic, *not* polylithic (a la LISP); there is no structure sharing.
- Lists are often useful for implementing a queue.

```
declare
   L : List;
   C : Cursor;
begin
   L.Append (New_Item => E);
   C := L.Last;

   E2 := Element (Position => C);
   Replace_Element (Position => C, By => E3);

   L.Insert (Before => C, New_Item => E4);
   L.Delete (Position => C);

   L.Delete_Last;
end;
```

Splice, Sort, Merge, etc

- Use Splice to either move an element (really, its node) within the same list or even from a different list, or to move an entire list.
- Like a vector, lists can be sorted. Unlike a vector, the sort is stable.
- A pair of sorted lists can be merged, such that one list is spliced onto the other in sort order.
- Operation Reverse_List reverses a list.
- There's a special Swap_Links for lists, to exchange list nodes instead of elements.

Associative Containers

- Associative containers (maps, sets) store elements ordered by key.
- Maps associate a separate key object with an element. For a set, an element is its own key.
- There are both ordered (tree-based) and hashed (hash table-based) versions.

Time Complexity

- Hashed associative containers have unit time complexity, on average. This is good for fast lookup of individual elements. (But note that execution behavior of a hashed container is very sensitive to quality of hash function.)
- Ordered associative containers have logarithmic time complexity, even in the worst case. This predictability is safer for real-time systems. Ordered containers are good for iteration over ranges of elements.

Hashed Container Equivalence

- A hashed container first computes the hash value of a new item, to find the bucket. It then uses the generic formal equivalence function (*not* equality) to compare the new item to the existing elements in that bucket. [See example on p.93.]

```
generic

   type Element_Type is private;

   with function Hash
     (Element : Element_Type) return Hash_Type;

   with function Equivalent_Elements
     (Left, Right : Element_Type) return Boolean;

   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
   -- ET."=" only used for Set."="; see p.74.

package Ada.Containers.Hashed_Sets is ...;
```

Ordered Container Equivalence

- During insertion in an ordered map or set, keys are compared for equivalence, not equality.
- Ordered keys are *equivalent* if the following relation (known as “strict weak ordering”) is true:

not (L < R) and not (R < L)

```
generic

   type Element_Type is private;

   with function "<" (Left, Right : Element_Type)
      return Boolean is <>;

   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
   -- ET."=" only used for Set."="; see p.75.

package Ada.Containers.Ordered_Sets is ...;
```

Maps

- Keys and elements are stored as pairs, ordered by key.
- For an ordered map, the “<“ relation for keys determines the (sorted) order.
- For a hashed map, the bucket (and hence the order) is determined by the hash value of the key. If the hash function is performing well, keys should be scattered throughout the hash table, equally distributed among the buckets.

Membership Tests

- The Contains and Find operations are used to determine whether an element is in the map. Use Contains if all you need is a simple membership test.
- Find returns a cursor as its result. If the cursor object has the distinguished value No_Element (or equivalently, the predicate Has_Element returns False), then the search failed and the key is not in the map. Otherwise, the cursor designates the key/element pair whose key matched.

Contains

```
M : Map;
...
procedure Op (Key : in Key_Type) is
begin
   if M.Contains (Key) then
      ... -- do something
   end if;
end Op;
```

Find

```
M : Map;
...
procedure Op (Key : in Key_Type) is
   procedure Process (K : KT; E : ET) is
   begin
      ... -- do something
   end;

   C : constant Cursor := M.Find (Key);
begin
   if Has_Element (C) then
      Query_Element (C, Process'Access);
   end if;
end Op;
```

Hashed Map "="

- For each key in the left map, hashed map equality searches for the key in the right map. That is, it first computes the hash value of the left key to find the right bucket, and then uses Equivalent_Keys to find the *equivalent* key in that bucket. If an equivalent key is found, it then compares elements using element equality. (This is the only time when element equality is actually used.)
- Note that both the key and the element are used to compute hashed map equality, but key equality is *not used*. (In fact key equality is *never* used in this API.)

```
generic

   type Key_Type is private;
   -- re-emergence of predefined "=" for KT is OK

   type Element_Type is private;

   with function Hash
     (Key : Key_Type) return Hash_Type;

   with function Equivalent_Keys
     (Left, Right : Key_Type) return Boolean;

   with function "=" (Left, Right : Element_Type)
     return Boolean is <>;
   -- need to pass in "=" for ET, since predefined
   -- equality is not necessarily what we want

package Ada.Containers.Hashed_Maps is ...;
```

Ordered Map "="

- Unlike a hashed map, there’s no need for ordered map "=" to search for the key, since the keys are already in sort order.
- If the key in the left map is *equivalent* to the corresponding key in the right map (equivalence being defined in terms of key "<"), then ordered map "=" compares the associated elements for equality. (This is the only time when element "=" is actually used.)
- Note that the key and element are both used to compute ordered map equality, but key equality is *not used*. (In fact key equality is *never* used in this API.)

```
generic

   type Key_Type is private;
   -- re-emergence of predefined "=" for KT is OK

   type Element_Type is private;

   with function "<" (Left, Right : Key_Type)
     return Boolean is <>;

   with function "=" (Left, Right : Element_Type)
     return Boolean is <>;
   -- need to pass in "=" for ET, since predefined
   -- equality is not necessarily what we want

package Ada.Containers.Ordered_Maps is ...;
```

Insertion

- The Insert operation attempts to insert a key (and element) into the map. If the key is already in the map, then Insert raises C_E; otherwise, it inserts the new key/element pair in the map.
- Include (a variation of Insert) attempts to insert the key, but if the key is already in the map, it replaces the existing key/element pair with the new key/element pair, instead of raising C_E.

Conditional Insertion

- Suppose we want to either insert a new element if this is a new key, or modify the existing element if the key already exists.
- One technique would be to first try to Find the key, and if it's not found, then Insert the key in the map.

```
Histogram : Map;  -- String key, Integer element
...
procedure Add (Word : in String) is
  procedure Increment_Count
    (Key : in String; Count : in out Integer) is
  begin
     Count := Count + 1;
  end;

  C : constant Cursor := Histogram.Find (Word);
begin
  if Has_Element (C) then  -- found
     Update_Element (C, Increment_Count'Access);
  else
     Histogram.Insert (Word, 1);
  end if;
end Add;
```

Condition Insertion (cont’d)

- However, this technique is inefficient, because Insert must perform its own search, thus duplicating the search performed by Find.
- A more efficient technique would be to attempt to insert the key, but instead of an exception, let the insertion operation return a cursor (the same as what Find does), and report back about whether the insertion succeeded.

```
procedure Op (M : in out Map) is
   C : Cursor;
   B : Boolean;
begin
   M.Insert   -- MORE INFO ON NEXT SLIDE
     (Key      => K,
      New_Item => E,
      Position => C,
      Inserted => B);

   if B then -- new key/elem inserted
      ...    -- C designates new key/elem
   else      -- key/elem not inserted
      ...    -- C designates existing key/elem
   end if;
end Op;
```

Conditional Insertion (cont'd)

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

```
Histogram : Map;
...
declare
  Position : Cursor;
  Inserted : Boolean;
begin
  Histogram.Insert
   (Key      => Word,
    New_Item => 0,         --yes: try to insert 0
    Position => Position,
    Inserted => Inserted); --result doesn't matter

  Update_Element (C, Increment_Count'Access);
end;
```

Deletion

- An element can be deleted either by specifying its key, or by specifying a cursor (that designates the key/element pair).
- The key-based Delete raises Constraint_Error if the key isn’t found in the map.
- Exclude (a variation of key-based Delete) does nothing if the key isn’t in the map.
- The cursor-based Delete raises C_E if the cursor equals No_Element, and raises Program_Error if the cursor designates a node in some other map.

```
procedure Op (M : in out Map) is
   C : Cursor;
   B : Boolean;
begin
   M.Insert (Key, E);
   M.Delete (Key);   -- by key

   M.Insert (Key, E, C, B);
   M.Delete (Position => C); -- by cursor

   M.Insert (Key, E);
   M.Exclude (Key); -- by key
end;
```

Replace, Replace_Element

- Replace searches the map to determine whether the key is a member. If the key isn't found, then it raises Constraint_Error. Otherwise, it replaces the existing key/element pair with the new key/element pair.
- Replace differs from Include only with respect to whether the key is already in the map.
- If you simply want to assign a new value to an existing element, then use Replace_Element.

Keys Get Updated Too

- A key might have interesting state of its own, and so Include and Replace assign new values to both the existing key and existing element.
- Sometimes the key assignment is more than you really want, if you're only interested in elements. You can avoid unwanted key assignment by using conditional Insert combined with Replace_Element.

```
M : Map;

procedure Op (K : KT; E : ET) is
begin
   M.Include (K, E);  -- replaces key too
end;

procedure Op2 (K : KT; E : ET) is
   C : Cursor;
   B : Boolean;
begin
   M.Insert (K, E, C, B);

   if not B then
      Replace_Element (C, By => E);
   end if;
end Op2;
```

Hashed Container Capacity

- As elements are inserted into the hashed container, the internal hash table automatically expands when it becomes full (defined as capacity = length).
- The standard does not specify what the load factor is. It says only that capacity is the maximum length before which no automatic rehashing will occur.
- **You need to care about rehashing, because it's expensive.** If you know the total number of elements prior to insertion, use Reserve_Capacity to preallocate the buckets array, and thus avoid rehashing.

```
procedure Op (N : Count_Type) is
  M : Map;
begin
  M.Reserve_Capacity (N);  -- Capacity >= N

  for I in Count_Type range 1 .. N loop
    M.Insert -- no resizing will occur
      (Key       => New_Key (I),
       New_Item  => New_Element (I));
  end loop;
  ...
end Op;
```

Sets

- A set is like a map, with the difference that in a set an element is its own key. There is no separate key object, and only the element is stored in the container.
- Ordered sets are often useful for implementing a priority queue.

```
procedure Op (S : in out Set) is
   C : Cursor;
   B : Boolean;
begin
   S.Insert (E);           -- can raise CE
   S.Insert (E2, C, B);    -- conditional
   S.Include (E3);         -- does not raise CE

   C := S.Find (E3);

   if Has_Element (C) then -- found
     ...
   end if;

   S.Delete (Item => E);
   S.Delete (Position => C);
   S.Exclude (Item => E2);
end Op;
```

Hashed Set "="

- Searches normally work by computing the hash value of the item to find the bucket, and then using Equivalent_Elements to find the matching element in the bucket.
- Hashed set equality works a little differently. It uses element equality ("=") to compare the item to the elements in the bucket. *This is the only time when element equality is used.*

Ordered Set "="

- Computing ordered set equality is straightforward: since the elements are already in (sorted) order, there's no need for a search. Each element in one set is simply compared to the corresponding element in the other set, using element equality. *This is the only time when element equality is used.*

Equivalent_Sets

- Interestingly, set containers actually have two ways of being compared. We have already seen the first way, set equality ("="), which is implemented in terms of element equality.
- The second way, Equivalent_Sets, is implemented in terms of the equivalence relation for elements. (Equivalent_Elements for hashed sets, and "<" for ordered sets.)

Classic Set Operations

- Set containers also have the traditional operations for sets: Union, Intersection, Difference, and Symmetric_Difference.
- Each operation has procedure, function, and operator forms.
- There are also Overlap and Is_Subset operations.

Generic_Keys

- The Generic_Keys nested package can be used to manipulate a set in terms of a key.
- Useful when the element is a record, and the element’s key is a component of the record.
- Solves the problem of finding a set element if you only know its key-part, and can't easily synthesize a nonce element to use as the search item.

```
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 Ada.Containers.Ordered_Sets
      (Employee_Type, "<");
```

```
Employees : Employee_Sets.Set;

procedure Add (SSN : in SSN_Type) is
   E : Employee_Type;
begin
   E.SSN := SSN;
   E.Name := ...;

   Employees.Insert (E);
end Op;
```

```
procedure Change_Address
   (SSN  : in SSN_Type;
    Home : in Address_Type) is

    C : Cursor := Employees.Find (Key => SSN); --?
    -- Find takes an Employee_Type, not an
    -- SSN_Type, so the code above won't compile.
begin
    ...
end Op;
```

```
function Get_SSN (E : Employee_Type) return SSN_Type is
begin
   return E.SSN;
end;

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

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

package SSN_Keys is new Employee_Sets.Generic_Keys
  (Key_Type => SSN_Type,
   Key      => Get_SSN,
   "<"      => "<",
   ">"      => ">");
```

```
procedure Change_Address
   (SSN  : in SSN_Type;
    Home : in Address_Type) is

    procedure Set_Home (E : in out Employee_Type) is
    begin
       E.Home := Home;  --benign change
    end;

    Position : constant Cursor :=
      SSN_Keys.Find (Employees, Key => SSN); --OK
begin
    if Has_Element (Position) then
       SSN_Keys.Update_Element_Preserving_Key
         (Container => Employees,
          Position  => Position,
          Process   => Set_Home'Access);
    end if;
end Op;
```

Miscellaneous

- Each container has different forms for definite and indefinite formal types. Useful when type String is the generic actual element or key type.
- The container library has generic operations for sorting both constrained and unconstrained arrays.
- There are also hash functions for String and Unbounded_String, and their wide string equivalents.

Word Frequency Example

- Suppose we are given the task of counting the frequency of each word in a file, and then displaying the results in frequency order.

Solution #1

- Instantiate an indefinite map (hashed or ordered -- it doesn't matter which) indexed by String. Use the map to collect the word frequencies.
- Allocate an array of map cursors, and then sort the array in frequency order.

```
with Ada.Containers.Indefinite_Hashed_Maps;
pragma Elaborate_All (Ada.Containers.Indefinite_Hashed_Maps);

with Ada.Strings.Hash;

package String_Integer_Maps is
  new Ada.Containers.Indefinite_Hashed_Maps
    (String,
     Integer,
     Ada.Strings.Hash,
     "=");

pragma Preelaborate (String_Integer_Maps);
```

```
M : String_Integer_Maps.Map;
...
procedure Insert (Word : String) is
  procedure Increment (K : String; E : in out Integer) is
     pragma Unreferenced (K);
  begin
     E := E + 1;  -- this is the count
  end;

  C : Cursor;
  B : Boolean;
begin
  M.Insert (Word, 0, C, B); -- yes, try to insert 0
  Update_Element (C, Increment'Access);
end Insert;
```

```ada
type Cursor_Array is
   array (Count_Type range <>) of String_Integer_Maps.Cursor;

A : Cursor_Array (1 .. M.Length);
...
Populate_Cursor_Array:
declare
   I : Count_Type := A'First;

   procedure Process (C : String_Integer_Maps.Cursor) is
   begin
     A (I) := C;
     I := I + 1;
   end;
begin
   M.Iterate (Process'Access);
end Populate_Cursor_Array;
```

```
Sort_Cursor_Array:
declare
   function "<" (L, R : String_Integer_Maps.Cursor) return Boolean is
      LE : constant Integer := Element (L);  -- L freq
      RE : constant Integer := Element (R);  -- R freq
   begin
      if LE = RE then                        -- counts match, so use word
         return Key (L) > Key (R);           -- order to break tie
      else
         return LE > RE;  -- sort in freq order
      end if;
   end "<";

   procedure Sort is new Ada.Containers.Generic_Array_Sort
     (Count_Type,
      String_Integer_Maps.Cursor,
      Cursor_Array,
      "<");
begin
   Sort (A);
end Sort_Cursor_Array;
```

```
Print_Cursor_Array:
for I in A'Range loop
   Put (Element (A (I)));  -- freq
   Put (' ');
   Put (Key (A (I)));  -- word
   New_Line;
end loop Print_Cursor_Array;
```

Solution #2

- Use a set with a word/count pair as the element.

```
type Data_Type (Word_Length : Positive) is record
    Word  : String (1 .. Word_Length);  -- key-part
    Count : Natural;                    -- payload
end record;

function Hash (Data : Data_Type) return Hash_Type is
begin
   return Ada.Strings.Hash (Data.Word); -- hash of key-part
end;

function Equivalent (L, R : Data_Type) return Boolean is
begin
   return L.Word = R.Word;  -- compare just key-parts
end;

package Data_Sets is new Ada.Containers.Indefinite_Hashed_Sets
  (Data_Type,
   Hash,
   Equivalent); -- ex. of why ET."=" isn't good enough
```

```
function Get_Word (Data : Data_Type) return String is
begin
   return Data.Word;
end;

function Equivalent (Word : String; Data : Data_Type)
  return Boolean is
begin
   return Word = Data.Word; -- compare key to key-part
end;

package Word_Keys is new Data_Sets.Generic_Keys
  (String,
   Get_Word,
   Ada.Strings.Hash,
   Equivalent);
```

```
Data_Set : in out Data_Sets.Set;

procedure Insert (Word : in String) is
  procedure Inc_Count (Data : in out Data_Type) is
  begin
     Data.Count := Data.Count + 1; -- update payload
  end;

  C : Data_Sets.Cursor;
  B : Boolean;
begin
  Data_Set.Insert ((Word'Length, Word, 0), C, B);

  Word_Keys.Update_Element_Preserving_Key
     (Data_Set, C, Inc_Count'Access);
end Insert;
```

Solution #3

- The indefinite vector has properties that make it an attractive alternative to sets and maps. It has less storage overhead, since there's no element node (just the element). And the elements are allocated, so the cost of insertion is relatively low (since only pointers to elements are moved, not elements).
- Here we use a binary search to find the insertion position such that the vector always remains sorted.
- A separate array of cursors, that we sort after collecting all the words, isn't necessary when using a vector, since we can explicitly sort the vector itself.

```
type Data_Type (Word_Length : Positive) is record
    Word  : String (1 .. Word_Length);  -- key-part
    Count : Natural;                    -- payload
end record;

package Data_Vectors is new Ada.Containers.Indefinite_Vectors
 (Positive,
  Data_Type);
```

```
Data_Vector : Data_Vectors.Vector;

procedure Insert (Word : in String) is

   I : Data_Vectors.Index_Subtype := Data_Vector.First_Index;
   J : Data_Vectors.Extended_Index := Data_Vector.Last_Index;
   K : Data_Vectors.Index_Subtype;
   Done : Boolean := False;

   procedure Process_K (Data : in out Data_Type) is
   begin
      if Data.Word < Word then
         I := K + 1;
      elsif Data.Word > Word then
         J := K - 1;
      else -- found equivalent word
         Data.Count := Data.Count + 1;  -- inc word count
         Done := True;
      end if;
   end Process_K;
...
```

```
 ...
begin -- Insert

   while I <= J loop  -- binary search

      K := I + (J - I) / 2;

      Data_Vector.Update_Element (K, Process_K'Access);

      if Done then
         return;
      end if;

   end loop;

   Data_Vector.Insert (I, Data_Type'(Word'Length, Word, 1));
   -- Word wasn't found: insert new word with count=1.

end Insert;
```

```
Sort_Data:
declare
   function "<" (L, R : Data_Type)
      return Boolean is
   begin
     if L.Count = R.Count then
        return L.Word > R.Word;
     else
        return L.Count > R.Count;
     end if;
   end "<";

   package Sorting is
      new Data_Vectors.Generic_Sorting ("<");
begin
   Sorting.Sort (Data_Vector);
end Sort_Data;
```