Tutorial: Charles Container Library

Ada-Europe 2004

Palma de Mallorca, Spain

presented by

Matthew Heaney

mailto:matthewjheaney@earthlink.net

http://home.earthlink.net/~matthewjheaney/

http://charles.tigris.org/

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, either 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, deques, 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 Hash_Type 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 Properties

- 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. It 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.
- For the range corresponding to all of the elements in a container, falling off the end (onto the Back 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

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

Stacks and Queues

- There are no “stack” or “queue” containers in Charles, because that functionality is already provided by the sequence containers.
- Stack functionality is provided by the vector, and queue functionality by the deque or list. A sorted set or map can be used as a priority queue.
- If you need to restrict access to only the end of the container, implement that feature yourself using a thin layer on top of vector, deque, or list.

Vectors

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

```
declare
   V : Vector_Subtype;
begin
   Append (V, New_Item => E);
   Delete_Last (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;
```

Vector Implementation

- Implemented internally as a contiguous array, with C convention.
- Function Size returns length of internal array.
- Use Resize to manually increase length of internal array; insertion is more efficient when expansion is done only once.

```
procedure Copy (A : Array_Subtype) is
   V : Vector_Types.Container_Type;
begin
   Resize (V, Size => A'Length);
   --If you know size prior to insertion,
   --resize first to avoid reallocation.

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

```ada
procedure Copy
   (A : in     Array_Subtype;
    V : in out Vector_Subtype;
    I : in     Index_Type'Base) is

    J : Index_Type'Base := I;
begin
    Insert_N (V, Before => I, Count => A'Length);
   -- dig the hole

    for Index in A'Range loop
       Replace_Element (V, J, By => A (Index));
       -- fill the hole

       J := J + 1;
    end loop;
    ...
end Copy;
```

```
procedure Process
   (E : in Element_Subtype) is ...;

procedure Op
   (V : in Vector_Subtype) is
begin
    for I in First (V) .. Last (V) loop
        Process (E => Element (V, I));
    end loop;
end Op;
```

Swap

- The internal arrays of two vectors can be exchanged using Swap.
- Swap is useful for *moving* a vector from one object to another, in contrast to assignment which *copies* the vector object.
- Swap is also useful for deallocating the internal array, as neither Clear nor Delete deallocates memory.

```
L : List_Of_Vectors.Container_Type;
...
declare
   V : Vector_Types.Container_Type;
   I : List_Of_Vectors.Iterator_Type;
begin
   Append (V, New_Item => E);
   ...      --populate V as appropriate
   Insert   --insert default-initialized element
     (Container => L,
      Before    => Back (L), --means append here
      Iterator  => I); --out param designates element

   declare
      V2 : Vector_Access renames To_Access (I).all;
   begin
      Swap (V, V2); --move, don't copy, vector object
   end;
end;
```

Bounded Vector

- Uses discriminant to specify size of stack-allocated internal array.
- Type isn't controlled, so it may be instantiated at any nesting level.
- Insertion raises exception if storage has already been exhausted.

```
Handles : Handle_Vectors.Container_Type (64);  -- Win max
...
Append (Handles, New_Item => H1);
Append (Handles, New_Item => H2);
...
Append (Handles, New_Item => H65); --Constaint_Error


type Handle_Access is access all Win32.HANDLE;
for Handle_Access'Storage_Size use 0;
pragma Convention (C, Handle_Access);

function To_Access is
  new Handle_Vectors.Generic_Element (Handle_Access);

WaitForMultipleObjects
 (Length (Handles),
  To_Access (Handles, Index => First (Handles)),
  False,
  INFINITE);
```

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 in middle slides elements towards the nearest end to make room for the new item(s).

```
declare
   D : Deque_Subtype;
begin
   Append (D, New_Item => E);
   Delete_Last (D);

   Prepend (D, New_Item => E);
   Delete_First (D);

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

Vector vs. Deque Storage

- A vector uses a contiguous array to store elements.
- A deque stores its elements on fixed-size blocks, and uses an offset to keep track of the first active element (on the first block).
- Prepend is only O(1) for a deque because it can simply decrement the offset of the first active element, and allocate a new block if necessary.

Vector vs. Deque Expansion

- Expansion in a vector works by allocating a new array, copying active elements from the old array onto the new array, and then deallocating the old array. (Note: Resize can be used to pre-allocate.)
- Expansion in a deque works by simply allocating a new block; there is no copying or deallocation.
- A deque is therefore potentially more efficient when the number of elements is large, and cannot be determined in advance of insertion.

Loops vs. Passive Iterators

- Both a vector and deque allow index-style iteration using a traditional for loop.
- However, if the container knows that it's visiting elements in sequence (as is the case in a passive iterator), then it can visit the elements in a way that takes advantage of that container's representation.

```
procedure Op (D : in Deque_Types.Container_Type) is
begin
   for I in First (D) .. Last (D) loop
     Do_Something (Element (D, I));
   end loop;
end;

--VERSUS--

procedure Op (D : in Deque_Types.Container_Type) is

   procedure Iterate is
     new Generic_Constant_Iteration (Do_Something);
begin
   Iterate (D);
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_Types.Container_Type;
   I : Iterator_Type;
begin
   Insert
     (Container => L,
      Before    => Back (L), --sentinel
      New_Item  => E,
      Iterator  => I);

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

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

Sentinel

- A list (and sets and maps) has a special sentinel node that is automatically allocated when the list object elaborates.
- The sentinel is designated by the iterator value returned by selector function Back.
- The sentinel has wrap-around semantics, meaning that the successor of Last is Back, and the predecessor of First is Back.

```
procedure Op (L : in List_Types.Container_Type) is

   I : Iterator_Type := First (L);
   J : constant Iterator_Type := Back (L);
begin
   while I /= J loop
      Process (Element (I));
      I := Succ (I);
   end loop;
end Op;
```

```
procedure Op (L : in List_Types.Container_Type) is

   I : Iterator_Type := Last (L);
   J : constant Iterator_Type := Back (L);
begin
   while I /= J loop
      Process (Element (I));
      I := Pred (I);
   end loop;
end Op;
```

Splice, Sort, and Merge

- Nodes in one list can be moved onto another list using Splice. Useful for implementing a holding area, e.g. a simple free store.
- Lists can sorted. The sort is stable.
- A pair of sorted lists can be merged, such that all the nodes one list are spliced onto another list in sort order.

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 List_Access is
   access all List_Subtype;

function To_Access is
   new Generic_Element (List_Access);

procedure Op (I : Iterator_Type) is
   L : List_Subtype renames
     To_Access (I).all;
begin
   Append (L, E); -- in-place modification
end;
```

Single Lists

- Internal storage nodes have only one link, to the next (successor) node.
- Only forward iteration is supported.
- The single list caches a pointer to the last node, so Append is only O(1) time complexity. This allows the single list to provide queue functionality, but with a lesser storage cost than a double list.

Double and Single Bounded Lists

- Maximum length is specified using a discriminant.
- Is not controlled, so it may be instantiated at any nesting level.

```
procedure Print (Histogram : in Map_Types.Container_Type) is

   package List_Types is  -- nested instantiation is allowed
      new Charles.Lists.Double.Bounded
        (Element_Type => Map_Types.Iterator_Type);

   List : List_Types.Container_Type (Size => Length (Histogram));

   procedure Process (I : in Map_Types.Iterator_Type) is
   begin
      Append (List, New_Item => I);
   end;

   procedure Populate_List is
     new Maps_Types.Generic_Iteration; -- use default name

begin -- Print

   Populate_List (Histogram);

   ... -- see next slide

end Print;
```

```
begin -- Print

   Populate_List (Histogram);

   Sort_List:
   declare
      function "<" (L, R : Map_Types.Iterator_Type)
         return Boolean is
      begin
         return Element (L) > Element (R); -- yes: count
      end;

      procedure Sort is
        new List_Types.Generic_Sort;  -- use "<" default
   begin
      Sort (List);
   end Sort_List;

   ... -- see next slide

end Print;
```

```
begin -- Print
   ...
   Print_Sorted_List:
   declare
      procedure Process  -- prints "n:word" to stdout
        (I : in List_Types.Iterator_Type) is
         J : Map_Types.Iterator_Type := Element (I);
      begin
         Put (Element (J), Width => 0);   -- the count
         Put (':');
         Put (Key (J));                   -- the word
         New_Line;
      end;

      procedure Print_Results is
        new List_Types.Generic_Iteration;
   begin
      Print_Results (List);
   end Print_Sorted_List;

end Print;
```

Associative Containers

- Associative containers (sets, maps) store elements ordered by key.
- There are both sorted (tree-based) and hashed (hash table-based) 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 “equivalent” 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.
- Sets have a nested generic, Generic_Keys, that allows you to perform key-based manipulation (Find, Delete, etc) of elements, very similar to a map.

Maps

- Elements are stored in key order (“<“ for sorted map, hash value for hashed map).
- 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;
```

Conditional Insertion

- In a map, keys are unique. If you attempt to insert a key already in the map, then the insertion will fail. How should this be reported?

Conditional Insertion

- One technique is to raise an exception. To avoid the exception (which indicates that a precondition has been violated), 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.

```
Histogram : Map_Types.Container_Type;
...
declare
  I : Iterator_Type := Find (Histogram, Word);
begin
  if I = Back (Histogram) then  -- not found
     Insert (Histogram, Word, 1);
  else
    declare
      N : Integer renames To_Access (I).all;
    begin
      N := N + 1;
    end;
  end if;
end;
```

Condition Insertion (cont'd)

- A more efficient technique is to attempt to insert the key, but let the insertion operation report whether the insertion was successful.
- Here the precondition is weaker, and so there is no exception if the key is already in the map.

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.

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

```
Histogram : Map_Types.Container_Type;
...
declare
  Iterator : Iterator_Type;
  Success  : Boolean;
begin
  Insert
   (Container => Histogram,
    Key       => Word,
    New_Item  => 0,          --yes: try to insert 0
    Iterator  => Iterator,
    Success   => Success); --result doesn't matter

  declare
    N : Integer renames To_Access (Iterator).all;
  begin
    N := N + 1; --inc result
  end;
end;
```

Replace

- Replace searches the map to determine whether the key is a member. If the key is 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.
- Similar to element assignment (see Replace_Element), but with the difference that a new key is created if it doesn't already exist.

```
Replace   --similar to M(K) := E;
 (Container => M,
  Key       => K,
  New_Item  => E);

--SAME AS:

declare
   I : Iterator_Type;
   B : Boolean;
begin
   Insert (M, K, E, I, B);

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

Hashed-Map Resize

- A hashed map is implemented using a hash table. As elements are inserted, the hash table expands when it becomes full, in order to preserve the load factor (α=1).
- As with a vector, if you know the total number of elements prior to insertion, use Resize to preallocate the buckets array.
- The buckets array is expanded to a length corresponding to a prime number. This produces better scatter when the hash value of the key is reduced modulo the size.

```
procedure Op (N : Natural) is
  Map : Map_Types.Container_Type;  -- Size = 0
begin
  Resize (Map, Size => N);  -- Size >= N

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

Deletion

- An element can be deleted either by specifying its key, or by specifying an iterator that designates the element.
- Deletion by iterator is probably more efficient, since there is no need to search for the key. (The iterator already designates the internal node of storage containing the key.)

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

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

```
procedure Finalize (Map : in out Map_Subtype) is
   I : Iterator_Type := First (Map);
   J : constant Iterator_Type := Back (Map);
begin
   while I /= J loop
      declare
         E : Element_Subtype renames
               To_Access (I).all;
      begin
         Finalize (E); -- or whatever
      end;

      Delete (Map, Iterator => I); --inc I
   end loop;
end Op;
```

Multimaps

- A multimap allows multiple keys to be equivalent (sorted) or equal (hashed).
- There is no conditional insert, because all insertions succeed.

```
procedure Op (M : in out Map_Subtype) is
   I : Iterator_Type;
begin
   Insert -- no success parameter needed
     (Container => M,
      Key       => K,
      New_Item  => E,
      Iterator  => I);
   ... -- I designates new key/element
end Op;
```

(Sorted) Equivalent Range

- Equivalent keys in a sorted multimap are contiguous, which means the range can be described using a half-open range iterator pair.
- Lower_Bound returns the smallest key in the map not less than a specified key.
- Upper_Bound returns the smallest key in the map greater than a specified key.

```
procedure Op (M : in Sorted_Map_Subtype) is
   I : Iterator_Type := Lower_Bound (M, K);
   J : Iterator_Type := Upper_Bound (M, K);
begin
   while I /= J loop
      declare
         E : Element_Subtype renames
            To_Access (I).all;
      begin
         ...
      end;

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

(Hashed) Equal Range

- Equal keys in a hashed multimap are stored contiguously in the same bucket.
- The simplest way to iterate over the range of equal keys is to use the passive iterator Generic_Equal_Range.

```
procedure Op (M : in Hashed_Map_Subtype) is

   procedure Process (I : Iterator_Type) is
      E : Element_Subtype := Element (I);
   begin
      ...
   end;

   procedure Iterate is
      new Generic_Equal_Range (Process);
begin
   Iterate (M, Key => K);
end Op;
```

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.

```
procedure Op (S : in out Set_Subtype) is
   I : Iterator_Type;
begin
   Insert (S, E);
   Insert (S, E2, I);

   I := Find (S, E3);

   if I /= Back (S) then -- found
     ...
   end if;

   Delete (S, Item => E);
   Delete (S, Iterator => I);
end Op;
```

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 an element if you only know its key.

```
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, "<");
```

```
Employees : Employee_Sets.Container_Type;

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

   Insert (S, E, I, Success => B);

   if not B then -- already in database
   ...
end Op;
```

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

    I : Iterator_Type :=
      Find (Employees, Key => SSN);  --?
begin
    ...
end Op;
```

```
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,
          "<"      => "<",
          ">"      => ">");
```

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

    I : Iterator_Type :=
      SSN_Keys.Find (Employees, Key => SSN);
begin
    if I /= Back (Employees) then
       declare
          E : Employee_Type renames
               To_Access (I).all;
       begin
          E.Home := Home; -- OK to modify
       ...                -- non-key part
end Op;
```

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.
- Lower_Bound and Upper_Bound can be used to describe the range of equivalent elements (in a sorted multiset).

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

   while I /= J loop ...;

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

Generic Algorithms

- An iterator pair specifies a "sequence of items," abstracting-away the actual container.
- We can write a generic algorithm strictly in terms of this *sequence view* to manipulate the items, thus allowing the algorithm to be used for any kind of container.

```
generic
   type Iterator_Type is private;

   with function Succ (Iterator : Iterator_Type)
     return Iterator_Type is <>;
   ...
procedure Generic_Algorithm
  (First, Back : in Iterator_Type);

pragma Pure (Generic_Algorithm);
```

Generic Algorithms

- Generic algorithms are implemented in terms of iterators only; this makes them neutral with respect to container.
- Generic algorithms are also agnostic with respect to elements. This allows the user to choose the most appropriate method to interrogate an element, given an iterator.

```
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 <>;

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

procedure Charles.Algorithms.Generic_Reverse_Bidirectional
  (First, Back : Iterator_Type);
```

```
procedure Charles.Algorithms.Generic_Reverse_Bidirectional
  (First, Back : Iterator_Type) is

   I : Iterator_Type := First;
   J : Iterator_Type := Back;

begin

   while I /= J loop

      J := Pred (J);

      exit when I = J;

      Swap (I, J);  --swap elements designated by I & J

      I := Succ (I);

   end loop;

end Charles.Algorithms.Generic_Reverse_Bidirectional;
```

```
procedure Reverse_Container (C : in CT) is

   procedure Swap (I, J : Iterator_Type) is
      E : Element_Type := Element (I);
   begin
      Replace_Element (I, By => Element (J));
      Replace_Element (J, By => E);
   end;

   procedure Reverse_Container is
      new Generic_Reverse_Bidirectional
         (Iterator_Type); --accept defaults
begin
   Reverse_Container (First (C), Back (C));
end;
```

```
procedure Reverse_Vector (V : in VT) is

   procedure Swap (I, J : Integer'Base) is
      E : Element_Type := Element (V, I);
   begin
      Replace_Element (V, I, By => Element (J));
      Replace_Element (V, J, By => E);
   end;

   procedure Reverse_Vector is
      new Generic_Reverse_Bidirectional
            (Iterator_Type => Integer'Base,
             Succ          => Integer'Succ,
             Pred          => Integer'Pred);
begin
   Reverse_Vector (First (V), Back (V));
end;
```

```
procedure Reverse_Array (A : in out Array_Type) is

   procedure Swap (I, J : Integer'Base) is
      E : constant Element_Type := A (I);
   begin
      A (I) := A (J);
      A (J) := E;
   end;

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

```
generic

   type Iterator_Type is private;

   with function Succ (I : Iterator_Type)
     return Iterator_Type is <>;

   with procedure Process (I : in Iterator_Type) is <>;

   with function Is_Less (L, R : Iterator_Type)
     return Boolean is <>;

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

procedure Charles.Algorithms.Generic_Set_Union
  (Left_First, Left_Back   : Iterator_Type;
   Right_First, Right_Back : Iterator_Type);
```

```
procedure Print_Union (S1, S2 : Set_Subtype) is

   procedure Process (I : Iterator_Type) is
   begin
      Put (Element (I)); Put (‘ ‘);
   end;

   function Is_Less (L, R : Iterator_Type)
     return Boolean is
   begin
      return Element (I) < Element (R);
   end;

   procedure Union is
      new Generic_Set_Union (Iterator_Type);
begin
   Union (First (S1), Back (S1), First (S2), Back (S2));
   New_Line;
end;
```

```
procedure Make_List_That_Is_Union_Of_Two_Sets
   (S1, S2 : in     Set_Subtype;
    L      : in out List_Subtype) is  --list of set iters

   procedure Process (I : Iterator_Type) is
   begin
      Append (L, New_Item => I); --store iter; means you
   end;                          --don't have to copy elem

   function Is_Less (L, R : Iterator_Type)
     return Boolean is
   begin
      return Element (I) < Element (R);
   end;

   procedure Union is
      new Generic_Set_Union (Iterator_Type);
begin
   Clear (L);
   Union (First (S1), Back (S1), First (S2), Back (S2));
end;
```

```
L1, L2 : List_Types.Container_Type;
... --populate L1 and L2
Sort (L1);  --instantiation of Generic_Sort
Sort (L2);

Make_Array_That_Is_Union_Of_Two_Sorted_Lists:
declare
   A : Array_Type (1 .. Length (L1) + Length (L2));
   I : Integer'Base := 1;

   procedure Process (Iter : Iterator_Type) is
   begin
      A (I) := Element (Iter);
      I := I + 1;
   end;

   function Is_Less (L, R : Iterator_Type) return Boolean is
   begin
      return Element (L) < Element (R);
   end;

   procedure Union is new Generic_Set_Union (Iterator_Type);
begin
   Union (First (L1), Back (L1), First (L2), Back (L2));
   ... --manipulate sorted array A
end Make_Array_That_Is_Union_Of_Two_Sorted_Lists;
```