Rationale for Ada 2005: 6a Containers

John Barnes

John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This paper describes the predefined container library in Ada 2005.

This is one of a number of papers concerning Ada 2005 which are being published in the Ada User Journal. An earlier version of this paper appeared in the Ada User Journal, Vol. 26, Number 4, December 2005. Other papers in this series will be found in other issues of the Journal or elsewhere on this website.

Keywords: rationale, Ada 2005.

1 Organization of containers

A major enhancement to the predefined library in Ada 2005 is the addition of a container library. This is quite extensive and merits this separate paper on its own. Other aspects of the predefined library and the overall rationale for extending the library were described in the previous paper.

The main packages in the container library can be grouped in various ways. One set of packages concerns the manipulation of objects of definite types and another, essentially identical, set concerns indefinite types. (Remember that an indefinite (sub)type is one for which we cannot declare an object without giving a constraint.) The reason for the duplication concerns efficiency. It is much easier to manipulate definite types and although the packages for indefinite types can be used for definite types, this would be rather inefficient.

We will generally only consider the definite packages. These in turn comprise two groups.

Sequence containers – these hold sequences of elements. There are packages for manipulating vectors and for manipulating linked lists. These two packages have much in common. But they have different behaviours in terms of efficiency according to the pattern of use. In general (with some planning) it should be possible to change from one to the other with little effort.

Associative containers – these associate a key with each element and then store the elements in order of the keys. There are packages for manipulating hashed maps, ordered maps, hashed sets and ordered sets. These four packages also have much in common and changing between hashed and ordered versions is usually feasible.

There are also quite separate generic procedures for sorting arrays which we will consider later.

The root package is

```
package Ada.Containers is
   pragma Pure(Containers);

   type Hash_Type is mod implementation-defined;
   type Count_Type is range 0 .. implementation-defined;

end Ada.Containers;
```

The type Hash_Type is used by the associative containers and Count_Type is used by both kinds of containers typically for the number of elements in a container. Note that we talk about elements in a container rather than the components in a container – components is the Ada term for the items of an array or record as an Ada type and it is convenient to use a different term since in the case of containers the actual data structure is hidden.

Worst-case and average-case time complexity bounds are given using the familiar *O*(...) notation. This encourages implementations to use techniques that scale reasonably well and avoid junk algorithms such as bubble sort.

Perhaps a remark about using containers from a multitasking program would be helpful. The general rule is given in paragraph 3 of Annex A which says "The implementation shall ensure that each language defined subprogram is reentrant in the sense that concurrent calls on the same subprogram perform as specified, so long as all parameters that could be passed by reference denote nonoverlapping objects." So in other words we have to protect ourselves by using the normal techniques such as protected objects when container operations are invoked concurrently on the same object from multiple tasks even if the operations are only reading from the container.

2 Lists and vectors

We will first consider the list container since in some ways it is the simplest. Here is its specification interspersed with some explanation

```
generic
  type Element_Type is private;
  with function "=" (Left, Right: Element_Type) return Boolean is <>;
package Ada.Containers.Doubly_Linked_Lists is
  pragma Preelaborate(Doubly_Linked_Lists);

  type List is tagged private;
  pragma Preelaborable_Initialization(List);
  type Cursor is private;
  pragma Preelaborable_Initialization(Cursor);
  Empty_List: constant List;
  No_Element: constant Cursor;
```

The two generic parameters are the type of the elements in the list and the definition of equality for comparing elements. This equality relation must be such that x = y and y = x always have the same value.

A list container is an object of the type List. It is tagged since it will inevitably be implemented as a controlled type. The fact that it is visibly tagged means that all the advantages of object oriented programming are available. For one thing it enables the use of the prefixed notation so that we can write operations such as

```
My_List.Append(Some_Value);
```

rather than

```
Append(My_List, Some_Value);
```

The type Cursor is an important concept. It provides the means of access to individual elements in the container. Not only does it contain a reference to an element but it also identifies the container as well. This enables various checks to be made to ensure that we don't accidentally meddle with an element in the wrong container.

The constants Empty_List and No_Element are as expected and also provide default values for objects of types List and Cursor respectively.

```
function "=" (Left, Right: List) return Boolean;
function Length(Container: List) return Count_Type;
function Is_Empty(Container: List) return Boolean;
procedure Clear(Container: in out List);
```

The function "=" compares two lists. It only returns true if both lists have the same number of elements and corresponding elements have the same value as determined by the generic parameter "=" for comparing elements. The subprograms Length, Is_Empty and Clear are as expected.

Note that A_List = Empty_List, Is_Empty(A_List) and Length(A_List) = 0 all have the same value.

```
function Element(Position: Cursor) return Element_Type;

procedure Replace_Element(Container: in out List; Position: in Cursor;
                          New_Item: in Element_Type);
```

These are the first operations we have met that use a cursor. The function Element takes a cursor and returns the value of the corresponding element (remember that a cursor identifies the list as well as the element itself). The procedure Replace_Element replaces the value of the element identified by the cursor by the value given; it makes a copy of course.

Note carefully that Replace_Element has both the list and cursor as parameters. There are two reasons for this concerning correctness. One is to enable a check that the cursor does indeed identify an element in the given list. The other is to ensure that we do have write access to the container (the parameter has mode **in out**). Otherwise it would be possible to modify a container even though we only had a constant view of it. So as a general principle any operation that modifies a container must have the container as a parameter whereas an operation that only reads it such as the function Element does not.

```
procedure Query_Element(Position: in Cursor;
                        Process: not null access procedure (Element: in Element_Type));

procedure Update_Element(Container: in out List; Position: in Cursor;
                         Process: not null access procedure (Element: in out Element_Type));
```

These procedures provide *in situ* access to an element. One parameter is the cursor identifying the element and another is an access to a procedure to be called with that element as parameter. In the case of Query_Element, we can only read the element whereas in the case of Update_Element we can change it as well since the parameter mode of the access procedure is **in out**. Note that Update_Element also has the container as a parameter for reasons just mentioned when discussing Replace_Element.

The reader might wonder whether there is any difference between calling the function Element to obtain the current value of an element and using the seemingly elaborate mechanism of Query_Element. The answer is that the function Element makes a copy of the value whereas Query_Element gives access to the value without making a copy. (And similarly for Replace_Element and Update_Element.) This wouldn't matter for a simple list of integers but it would matter if the elements were large or of a controlled type (maybe even lists themselves).

```
procedure Move(Target, Source: in out List);
```

This moves the list from the source to the target after first clearing the target. It does not make copies of the elements so that after the operation the source is empty and Length(Source) is zero.

```
procedure Insert(Container: in out List;
                 Before: in Cursor;
                 New_Item: in Element_Type;
                 Count: in Count_Type := 1);
```

```
procedure Insert(Container: in out List;
                 Before: in Cursor;
                 New_Item: in Element_Type;
                 Position: out Cursor;
                 Count: in Count_Type := 1);

procedure Insert(Container: in out List;
                 Before: in Cursor;
                 Position: out Cursor;
                 Count: in Count_Type := 1);
```

These three procedures enable one or more identical elements to be added anywhere in a list. The place is indicated by the parameter Before – if this is No_Element, then the new elements are added at the end. The second procedure is similar to the first but also returns a cursor to the first of the added elements. The third is like the second but the new elements take their default values. Note the default value of one for the number of elements.

```
procedure Prepend(Container: in out List;
                  New_Item: in Element_Type;
                  Count: in Count_Type := 1);

procedure Append(Container: in out List;
                 New_Item: in Element_Type;
                 Count: in Count_Type := 1);
```

These add one or more new elements at the beginning or end of a list respectively. Clearly these operations can be done using Insert but they are sufficiently commonly needed that it is convenient to provide them specially.

```
procedure Delete(Container: in out List;
                 Position: in out Cursor;
                 Count: in Count_Type := 1);

procedure Delete_First(Container: in out List; Count: in Count_Type := 1);

procedure Delete_Last(Container: in out List; Count: in Count_Type := 1);
```

These delete one or more elements at the appropriate position. In the case of Delete, the parameter Position is set to No_Element upon return. If there are not as many as Count elements to be deleted at the appropriate place then it just deletes as many as possible (this clearly results in the container becoming empty in the case of Delete_First and Delete_Last).

```
procedure Reverse_Elements(Container: in out List);
```

This does the obvious thing. It would have been nice to call this procedure Reverse but sadly that is a reserved word.

```
procedure Swap(Container: in out List; I, J: in Cursor);
```

This handy procedure swaps the values in the two elements denoted by the two cursors. The elements must be in the given container otherwise Program_Error is raised. Note that the cursors do not change.

```
procedure Swap_Links(Container: in out List; I, J: in Cursor);
```

This performs the low level operation of swapping the links rather than the values which can be much faster if the elements are large. There is no analogy in the vectors package.

```
procedure Splice(Target: in out List;
                 Before: in Cursor;
                 Source in out List);

procedure Splice(Target: in out List;
                 Before: in Cursor;
                 Source: in out List;
                 Position: in out Cursor);

procedure Splice(Container: in out List;
                 Before: in Cursor;
                 Position: in out Cursor);
```

These three procedures enable elements to be moved (without copying). The place is indicated by the parameter Before – if this is No_Element, then the elements are added at the end. The first moves all the elements of Source into Target at the position given by Before; as a consequence, like the procedure Move, after the operation the source is empty and Length(Source) is zero. The second moves a single element at Position from the list Source to Target and so the length of target is incremented whereas that of source is decremented; Position is updated to its new location in Target. The third moves a single element within a list and so the length remains the same (note the formal parameter is Container rather than Target in this case). There are no corresponding operations in the vectors package because, like Swap_Links, we are just moving the links and not copying the elements.

```
function First(Container: List) return Cursor;
function First_Element(Container: List) return Element_Type;
function Last(Container: List) return Cursor;
function Last_Element(Container: List) return Element_Type;
function Next(Position: Cursor) return Cursor;
function Previous(Position: Cursor) return Cursor;
procedure Next(Position: in out Cursor);
procedure Previous(Position: in out Cursor);

function Find(Container: List;
              Item: Element_Type;
              Position: Cursor:= No_Element) return Cursor;

function Reverse_Find(Container: List;
              Item: Element_Type;
              Position: Cursor:= No_Element) return Cursor;

function Contains(Container: List; Item: Element_Type) return Boolean;
```

Hopefully the purpose of these is almost self-evident. The function Find searches for an element with the given value starting at the given cursor position (or at the beginning if the position is No_Element); if no element is found then it returns No_Element. Reverse_Find does the same but backwards. Note that equality used for the comparison in Find and Reverse_Find is that defined by the generic parameter "=".

```
function Has_Element(Position: Cursor) return Boolean;
```

This returns False if the cursor does not identify an element; for example if it is No_Element.

```
procedure Iterate(Container: in List;
              Process: not null access procedure (Position: in Cursor));

procedure Reverse_Iterate(Container: in List;
              Process: not null access procedure (Position: in Cursor));
```

These apply the procedure designated by the parameter Process to each element of the container in turn in the appropriate order.

```
generic
   with function "<" (Left, Right: Element_Type) return Boolean is <>;
package Generic_Sorting is
   function Is_Sorted(Container: List) return Boolean;
   procedure Sort(Container: in out List);
   procedure Merge(Target, Source: in out List);
end Generic_Sorting;
```

This generic package performs sort and merge operations using the order specified by the generic formal parameter. Note that we use generics rather than access to subprogram parameters when the formal process is given by an operator. This is because the predefined operations have convention Intrinsic and one cannot pass an intrinsic operation as an access to subprogram parameter. The function Is_Sorted returns True if the container is already sorted. The procedure Sort arranges the elements into order as necessary – note that no copying is involved since it is only the links that are moved. The procedure Merge takes the elements from Source and adds them to Target. After the merge Length(Source) is zero. If both lists were sorted before the merge then the result is also sorted.

And finally we have

```
private
   ... -- not specified by the language
end Ada.Containers.Doubly_Linked_Lists;
```

If the reader has got this far they have probably understood how to use this package so extensive examples are unnecessary. However, as a taste, here is a simple stack of floating point numbers

```
package Stack is
   procedure Push(X: in Float);
   function Pop return Float;
   function Size return Integer;
   exception Stack_Empty;
end;

with Ada.Containers.Doubly_Linked_Lists;
use Ada.Containers;
package body Stack is

   package Float_Container is new Doubly_Linked_Lists(Float);
   use Float_Container;
   The_Stack: List;

   procedure Push(X: in Float) is
   begin
      Append(The_Stack, X);                    -- or The_Stack.Append(X);
   end Push;

   function Pop return Float is
      Result: Float;
   begin
      if Is_Empty(The_Stack) then
         raise Stack_Empty;
      end if;
      Result := Last_Element(The_Stack);
      Delete_Last(The_Stack);
      return Result;
    end Pop;

    function Size return Integer is
    begin
      return Integer(Length(The_Stack));
    end Size;

  end Stack;
```

This barely needs any explanation. The lists package is instantiated in the package Stack and the object The_Stack is of course the list container. The rest is really straightforward. We could of course use the prefixed notation throughout as indicated in Push.

An important point should be mentioned concerning lists (and containers in general). This is that attempts to do foolish things typically result in Constraint_Error or Program_Error being raised. This especially applies to the procedures Process in Query_Element, Update_Element, Iterate and Reverse_Iterate. The concepts of tampering with cursors and elements are introduced in order to dignify a general motto of "Thou shalt not violate thy container".

Tampering with cursors occurs when elements are added to or deleted from a container (by calling Insert and so on) whereas tampering with elements means replacing an element (by calling Replace_Element for example). Tampering with elements is a greater sin and includes tampering with cursors. The procedure Process in Query_Element and Update_Element must not tamper with elements and the procedure Process in the other cases must not tamper with cursors. The reader might think it rather odd that Update_Element should not be allowed to tamper with elements since the whole purpose is to update the element; this comes back to the point mentioned earlier that update element gives access to the existing element *in situ* via the parameter of Process and that is allowed – calling Replace_Element within Process would be tampering. Tampering causes Program_Error to be raised.

We will now consider the vectors package. Its specification starts

```
generic
  type Index_Type is range <>;
  type Element_Type is private;
  with function "=" (Left, Right: Element_Type) return Boolean is <>;
package Ada.Containers.Vectors is
  pragma Preelaborate(Vectors);
```

This is similar to the lists package except for the additional generic parameter Index_Type (note that this is an integer type and not a discrete type). This additional parameter reflects the idea that a vector is essentially an array and we can index directly into an array.

In fact the vectors package enables us to access elements either by using an index or by using a cursor. Thus many operations are duplicated such as

```
function Element(Container: Vector; Index: Index_Type) return Element_Type;
function Element(Position: Cursor) return Element_Type;

procedure Replace_Element(Container: in out Vector;
                          Index: in Index_Type;
                          New_Item: in Element_Type);

procedure Replace_Element(Container: in out Vector;
                          Position: in Cursor;
                          New_Item: in Element_Type);
```

If we use an index then there is always a distinct parameter identifying the vector as well. If we use a cursor then the vector parameter is omitted if the vector is unchanged as is the case with the function Element. Remember that we stated earlier that a cursor identifies both an element and the container but if the container is being changed as in the case of Replace_Element then the container has to be passed as well to ensure write access and to enable a check that the cursor does identify an element in the correct container.

There are also functions First_Index and Last_Index thus

```
function First_Index(Container: Vector) return Index_Type;

function Last_Index(Container: Vector) return Extended_Index;
```

These return the values of the index of the first and last elements respectively. The function First_Index always returns Index_Type'First whereas Last_Index will return No_Index if the vector is empty. The function Length returns Last_Index–First_Index+1 which is zero if the vector is empty. Note that the irritating subtype Extended_Index has to be introduced in order to cope with end values. The constant No_Index has the value Extended_Index'First which is equal to Index_Type'First–1.

There are operations to convert between an index and a cursor thus

```
function To_Cursor(Container: Vector; Index: Extended_Index) return Cursor;

function To_Index(Position: Cursor) return Extended_Index;
```

It is perhaps slightly messier to use the index and vector parameters because of questions concerning the range of values of the index but probably slightly faster and maybe more familiar. And sometimes of course using an index is the whole essence of the problem. In the paper on access types we showed a use of the procedure Update_Element to double the values of those elements of a vector whose index was in the range 5 to 10. This would be tedious with cursors.

But an advantage of using cursors is that (provided certain operations are avoided) it is easy to replace the use of vectors by lists.

For example here is the stack package rewritten to use vectors

```
with Ada.Containers.Vectors;                          -- changed
use Ada.Containers;
package body Stack is

  package Float_Container is new Vectors(Natural, Float); -- changed
  use Float_Container;
  The_Stack: Vector;                                  -- changed

  procedure Push(X: in Float) is
  begin
    Append(The_Stack, X);
  end Push;

  -- etc exactly as before

end Stack;
```

So the changes are very few indeed and can be quickly done with a simple edit.

Note that the index parameter has been given as Natural rather than Integer. Using Integer will not work since attempting to elaborate the subtype Extended_Index would raise Constraint_Error when evaluating Integer'First–1. But in any event it is more natural for the index range of the container to start at 0 (or 1) rather than a large negative value such as Integer'First.

There are other important properties of vectors that should be mentioned. One is that there is a concept of capacity. Vectors are adjustable and will extend if necessary when new items are added. However, this might lead to lots of extensions and copying and so we can set the capacity of a container by calling

```
procedure Reserve_Capacity(Container: in out Vector; Capacity: in Count_Type);
```

There is also

```
function Capacity(Container: Vector) return Count_Type;
```

which naturally returns the current capacity. Note that Length(V) cannot exceed Capacity(V) but might be much less.

If we add items to a vector whose length and capacity are the same then no harm is done. The capacity will be expanded automatically by effectively calling Reserve_Capacity internally. So the user does not need to set the capacity although not doing so might result in poorer performance.

There is also the concept of "empty elements". These are elements whose values have not been set. There is no corresponding concept with lists. It is a bounded error to read an empty element. Empty elements arise if we declare a vector by calling

```
function To_Vector(Length: Count_Type) return Vector;
```

as in

```
My_Vector: Vector := To_Vector(100);
```

There is also the much safer

```
function To_Vector(New_Item: Element_Type; Length: Count_Type) return Vector;
```

which sets all the elements to the value New_Item.

There is also a procedure

```
procedure Set_Length(Container: in out Vector; Length: in Count_Type);
```

This changes the length of a vector. This may require elements to be deleted (from the end) or to be added (in which case the new ones are empty).

The final way to get an empty element is by calling one of

```
procedure Insert_Space(Container: in out Vector;
                       Before: in Extended_Index;
                       Count: in Count_Type := 1);

procedure Insert_Space(Container: in out Vector;
                       Before: in Cursor;
                       Position: out Cursor;
                       Count: in Count_Type := 1);
```

These insert the number of empty elements given by Count at the place indicated. Existing elements are slid along as necessary. These should not be confused with the versions of Insert which do not provide an explicit value for the elements – in those cases the new elements take their default values.

Care needs to be taken if we use empty elements. For example we should not compare two vectors using "=" if they have empty elements because this implies reading them. But the big advantage of empty elements is that they provide a quick way to make a large lump of space in a vector which can then be filled in with appropriate values. One big slide is a lot faster than lots of little ones.

For completeness, we briefly mention the remaining few subprograms that are unique to the vectors package.

There are further versions of Insert thus

```
procedure Insert(Container: in out Vector;
                  Before: in Extended_Index; New_Item: in Vector);

procedure Insert(Container: in out Vector;
                  Before: in Cursor; New_Item: in Vector);

procedure Insert(Container: in out Vector;
                  Before: in Cursor; New_Item: in Vector; Position: out Cursor);
```

These insert copies of a vector into another vector (rather than just single elements).

There are also corresponding versions of Prepend and Append thus

```
procedure Prepend(Container: in out Vector; New_Item: in Vector);

procedure Append(Container: in out Vector; New_Item: in Vector);
```

Finally, there are four functions "&" which concatenate vectors and elements by analogy with those for the type String. Their specifications are

```
function "&" (Left, Right: Vector) return Vector;
function "&" (Left: Vector; Right: Element_Type) return Vector;
function "&" (Left: Element_Type; Right: Vector) return Vector;
function "&" (Left, Right: Element_Type) return Vector;
```

Note the similarity between

```
Append(V1, V2);
V1 := V1 & V2;
```

The result is the same but using "&" is less efficient because of the extra copying involved. But "&" is a familiar operation and so is provided for convenience.

3 Maps

We will now turn to the maps and sets packages. We will start by considering maps which are more exciting than sets and begin with ordered maps which are a little simpler and then consider hashed maps.

Remember that a map is just a means of getting from a value of one type (the key) to another type (the element). This is not a one-one relationship. Given a key there is a unique element (if any), but several keys may correspond to the same element. A simple example is an array. This is a map from the index type to the component type. Thus if we have

```
S: String := "animal";
```

then this provides a map from integers in the range 1 to 6 to some values of the type Character. Given an integer such as 3 there is a unique character 'i' but given a character such as 'a' there might be several corresponding integers (in this case both 1 and 5).

More interesting examples are where the set of used key values is quite sparse. For example we might have a store where various spare parts are held. The parts have a five-digit part number and there are perhaps twenty racks where they are held identified by a letter. However, only a handful of the five digit numbers are in use so it would be very wasteful to use an array with the part number as index. What we want instead is a container which holds just the pairs that matter such as (34618, 'F'), (27134, 'C') and so on. We can do this using a map. We usually refer to the pairs of values as nodes of the map.

There are two maps packages with much in common. One keeps the keys in order and the other uses a hash function. Here is the specification of the ordered maps package generally showing just those facilities common to both.

```
generic
   type Key_Type is private;
   type Element_Type is private;
   with function "<" (Left, Right: Key_Type) return Boolean is <>;
   with function "=" (Left, Right: Element_Type) return Boolean is <>;
package Ada.Containers.Ordered_Maps is
   pragma Preelaborate(Ordered_Maps);

   function Equivalent_Keys(Left: Right: Key_Type) return Boolean;
```

The generic parameters include the ordering relationship "<" on the keys and equality for the elements.

It is assumed that the ordering relationship is well behaved in the sense that if x < y is true then y < x is false. We say that two keys x and y are equivalent if both x < y and y < x are false. In other words this defines an equivalence class on keys. The relationship must also be transitive, that is, if x < y and y < z are both true then x < z must also be true.

This concept of an equivalence relationship occurs throughout the various maps and sets. Sometimes, as here, it is defined in terms of an order but in other cases, as we shall see, it is defined by an equivalence function.

It is absolutely vital that the equivalence relations are defined properly and meet the above requirements. It is not possible for the container packages to check this and if the operations are wrong then peculiar behaviour is almost inevitable.

For the convenience of the user the function Equivalent_Keys is declared explicitly. It is equivalent to

```
function Equivalent_Keys(Left, Right: Key_Type) return Boolean is
begin
   return not (Left < Right) and not (Right < Left);
end Equivalent_Keys;
```

The equality operation on elements is not so demanding. It must be symmetric so that x = y and y = x are the same but transitivity is not required (although cases where it would not automatically be transitive are likely to be rare). The operation is only used for the function "=" on the containers as a whole.

Note that Find and similar operations for maps and sets work in terms of the equivalence relationship rather than equality as was the case with lists and vectors.

```
   type Map is tagged private;
   pragma Preelaborable_Initialization(Map);
   type Cursor is private;
   pragma Preelaborable_Initialization(Cursor);
   Empty_Map: constant Map;
   No_Element: constant Cursor;
```

The types Map and Cursor and constants Empty_Map and No_Element are similar to the corresponding entities in the lists and vectors containers.

```
function "=" (Left, Right: Map) return Boolean;
function Length(Container: Map) return Count_Type;
function Is_Empty(Container: Map) return Boolean;
procedure Clear(Container: in out Map);
```

These are again similar to the corresponding entities for lists. Note that two maps are said to be equal if they have the same number of nodes with equivalent keys (as defined by "<") whose corresponding elements are equal (as defined by "=").

```
function Key(Position: Cursor) return Key_Type;
function Element(Position: Cursor) return Element_Type;

procedure Replace_Element(Container: in out Map;
                          Position: in Cursor;
                          New_Item: in Element_Type);

procedure Query_Element(Position: in Cursor;
   Process: not null access procedure (Key: in Key_Type; Element: in Element_Type));

procedure Update_Element(Container: in out Map; Position: in Cursor;
   Process: not null access procedure (Key: in Key_Type; Element: in out Element_Type));
```

In this case there is a function Key as well as a function Element. But there is no procedure Replace_Key since it would not make sense to change a key without changing the element as well and this really comes down to deleting the whole node and then inserting a new one.

The procedures Query_Element and Update_Element are slightly different in that the procedure Process also takes the key as parameter as well as the element to be read or updated. Note again that the key cannot be changed. Nevertheless the value of the key is given since it might be useful in deciding how the update should be performed. Remember that we cannot get uniquely from an element to a key but only from a key to an element.

```
procedure Move(Target, Source: in out Map);
```

This moves the map from the source to the target after first clearing the target. It does not make copies of the nodes so that after the operation the source is empty and Length(Source) is zero.

```
procedure Insert(Container: in out Map;
                 Key: in Key_Type;
                 New_Item: in Element_Type;
                 Position: out Cursor;
                 Inserted: out Boolean);

procedure Insert(Container: in out Map;
                 Key: in Key_Type;
                 Position: out Cursor;
                 Inserted: out Boolean);

procedure Insert(Container: in out Map;
                 Key: in Key_Type;
                 New_Item: in Element_Type);
```

These insert a new node into the map unless a node with an equivalent key already exists. If it does exist then the first two return with Inserted set to False and Position indicating the node whereas the third raises Constraint_Error (the element value is not changed). If a node with equivalent key is not found then a new node is created with the given key, the element value is set to New_Item when that is given and otherwise it takes its default value (if any), and Position is set when given.

Unlike vectors and lists, we do not have to say where the new node is to be inserted because of course this is an ordered map and it just goes in the correct place according to the order given by the generic parameter "<".

```
procedure Include(Container: in out Map;
                  Key: in Key_Type;
                  New_Item: in Element_Type);
```

This is somewhat like the last Insert except that if an existing node with an equivalent key is found then it is replaced (rather than raising Constraint_Error). Note that both the key and the element are updated. This is because equivalent keys might not be totally equal.

For example the key part might be a record with part number and year of introduction, thus

```
type Part_Key is
  record
    Part_Number: Integer;
    Year: Integer;
  end record;
```

and we might define the ordering relationship to be used as the generic parameter simply in terms of the part number

```
function "<" (Left, Right: Part_Key) return Boolean is
begin
  return Left.Part_Number < Right.Part_Number;
end "<";
```

In this situation, the keys could match without the year component being the same and so it would need to be updated. In other words with this definition of the ordering relation, two keys are equivalent provided just the part numbers are the same.

```
procedure Replace(Container: in out Map;
                  Key: in Key_Type;
                  New_Item: in Element_Type);
```

In this case, Constraint_Error is raised if the node does not already exist. On replacement both the key and the element are updated as for Include.

Perhaps a better example of equivalent keys not being totally equal is if the key were a string. We might decide that the case of letter did not need to match in the test for equivalence but nevertheless we would probably want to update with the string as used in the parameter of Replace.

```
procedure Exclude(Container: in out Map; Key: in Key_Type);
```

If there is a node with an equivalent key then it is deleted. If there is not then nothing happens.

```
procedure Delete(Container: in out Map; Key: in Key_Type);

procedure Delete(Container: in out Map; Position: in out Cursor);
```

These delete a node. In the first case if there is no such equivalent key then Constraint_Error is raised (by contrast to Exclude which remains silent in this case). In the second case if the cursor is No_Element then again Constraint_Error is raised – there is also a check to ensure that the cursor otherwise does designate a node in the correct map (remember that cursors designate both an entity and the container); if this check fails then Program_Error is raised.

Perhaps it is worth observing that Insert, Include, Replace, Exclude and Delete form a sort of progression from an operation that will insert something, through operations that might insert, will neither insert nor delete, might delete, to the final operation that will delete something. Note also that Include, Replace and Exclude do not apply to lists and vectors.

```
function First(Container: Map) return Cursor;
function Last(Container: Map) return Cursor;
function Next(Position: Cursor) return Cursor;
procedure Next(Position: in out Cursor);
function Find(Container: Map; Key: Key_Type) return Cursor;
function Element(Container: Map; Key: Key_Type) return Element;
function Contains(Container: Map; Key: Key_Type) return Boolean;
```

These should be self-evident. Unlike the operations on vectors and lists, Find logically searches the whole map and not just starting at some point (and since it searches the whole map there is no point in having Reverse_Find). (In implementation terms it won't actually search the whole map because it will be structured in a way that makes this unnecessary – as a balanced tree perhaps.) Moreover, Find uses the equivalence relation based on the "<" parameter so in the example it only has to match the part number and not the year. The function call Element(My_Map, My_Key) is equivalent to Element(Find(My_Map, My_Key)).

```
function Has_Element(Position: Cursor) return Boolean;

procedure Iterate(Container: in Map;
            Process: not null access procedure (Position: in Cursor));
```

These are also as for other containers.

And at last we have

```
private
  ... -- not specified by the language
end Ada.Containers.Ordered_Maps;
```

We have omitted to mention quite a few operations that have no equivalent in hashed maps – we will come back to these in a moment.

As an example we can make a container to hold the information concerning spare parts. We can use the type Part_Key and the function "<" as above. We can suppose that the element type is

```
type Stock_Info is
  record
    Shelf: Character range 'A' .. 'T';
    Stock: Integer;
  end record;
```

This gives both the shelf letter and the number in stock.

We can then declare the container thus

```
package Store_Maps is
   new Ordered_Maps(Key_Type => Part_Key,
                    Element_Type => Stock_Info,
                    "<" => "<");

The_Store: Store_Maps.Map;
```

The last parameter could be omitted because the formal has a <> default.

We can now add items to our store by calling

```
The_Store.Insert((34618, 1998), ('F', 25));
The_Store.Insert((27134, 2004), ('C', 45));
...
```

We might now have a procedure which, given a part number, checks to see if it exists and that the stock is not zero, and if so returns the shelf letter and year number and decrements the stock count.

```
procedure Request(Part: in Integer; OK: out Boolean;
                    Year: out Integer; Shelf: out Character) is
  C: Cursor;
  K: Part_Key;
  E: Stock_Info;
begin
  C := The_Store.Find((Part, 0));
  if C = No_Element then
    OK := False; return;                -- no such key
  end if;
  E := Element(C);  K := Key(C);
  Year := K.Year;  Shelf := E.Shelf;
  if E.Stock = 0 then
    OK := False; return;                -- out of stock
  end if;
  Replace_Element(C, (Shelf, E.Stock–1));
  OK := True;
end Request;
```

Note that we had to put a dummy year number in the call of Find. We could of course use the new <> notation for this

```
C := The_Store.Find((Part, others => <>));
```

The reader can improve this example at leisure – by using Update_Element for example.

As another example suppose we wish to check all through the stock looking for parts whose stock is low, perhaps less than some given parameter. We can use Iterate for this as follows

```
procedure Check_Stock(Low: in Integer) is

  procedure Check_It(C: in Cursor) is
  begin
    if Element(C).Stock < Low then
      -- print a message perhaps
      Put("Low stock of part ");
      Put_Line(Key(C).Part_Number);
    end if;
  end Check_It;

begin
  The_Store.Iterate(Check_It'Access);
end Check_Stock;
```

Note that this uses a so-called downward closure. The procedure Check_It has to be declared locally to Check_Stock in order to access the parameter Low. (Well you could declare it outside and copy the parameter Low to a global variable but that is just the sort of wicked thing one has to do in lesser languages (such as even Ada 95). It is not task safe for one thing.)

Another approach is to use First and Next and so on thus

```
procedure Check_Stock(Low: in Integer) is
  C: Cursor := The_Store.First;
begin
  loop
    exit when C = No_Element;
    if Element(C).Stock < Low then
      -- print a message perhaps
      Put("Low stock of part ");
      Put_Line(Key(C).Part_Number);
    end if;
    C := The_Store.Next(C);
  end loop;
end Check_Stock;
```

We will now consider hashed maps. The trouble with ordered maps in general is that searching can be slow when the map has many entries. Techniques such as a binary tree can be used but even so the search time will increase at least as the logarithm of the number of entries. A better approach is to use a hash function. This will be familiar to many readers (especially those who have written compilers). The general idea is as follows.

We define a function which takes a key and returns some value in a given range. In the case of the Ada containers it has to return a value of the modular type Hash_Type which is declared in the root package Ada.Containers. We could then convert this value onto a range representing an index into an array whose size corresponds to the capacity of the map. This index value is the preferred place to store the entry. If there already is an entry at this place (because some other key has hashed to the same value) then a number of approaches are possible. One way is to create a list of entries with the same index value (often called buckets); another way is simply to put it in the next available slot. The details don't matter. But the overall effect is that provided the map is not too full and the hash function is good then we can find an entry almost immediately more or less irrespective of the size of the map.

So as users all we have to do is to define a suitable hash function. It should give a good spread of values across the range of Hash_Type for the population of keys, it should avoid clustering and above all for a given key it must *always* return the same hash value. A good discussion on hash functions by Knuth will be found in [1].

Defining good hash functions needs care. In the case of the part numbers we might multiply the part number by some obscure prime number and then truncate the result down to the modular type Hash_Type. The author hesitates to give an example but perhaps

```
function Part_Hash(P: Part_Key) return Hash_Type is
  M31: constant := 2**31–1;                    -- a nice Mersenne prime
begin
  return Hash_Type(P.Part_Number) * M31;
end Part_Hash;
```

On reflection that's probably a very bad prime to use because it is so close to half of 2**32 a typical value of Hash_Type'Last+1. Of course it doesn't have to be prime but simply relatively prime to it such as 5**13. Knuth suggests dividing the range by the golden number $\tau = (\sqrt{5}+1)/2 = 1.618...$ and then taking the nearest number relatively prime which is in fact simply the nearest odd number (in this case it is 2654435769).

Here is a historic interlude. Marin Mersenne (1588-1648) was a Franciscan monk who lived in Paris. He studied numbers of the form $M_p = 2^p - 1$ where p is prime. A lot of these are themselves prime. Mersenne gave a list of those upto 257 which he said were prime (namely 2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257). It was not until 1947 that it was finally settled that he got some of them wrong (61, 89, and 107 are also prime but 67 and 257 are not). At the time of writing there are 42 known Mersenne primes and the largest which is also the largest known prime number is $M_{25964951}$ – see www.mersenne.org.

The specification of the hashed maps package is very similar to that for ordered maps. It starts

```
generic
   type Key_Type is private;
   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 <>;
package Ada.Containers.Hashed_Maps is
   pragma Preelaborate(Hashed_Maps);
```

The differences from the ordered maps package are that there is an extra generic parameter Hash giving the hash function and the ordering parameter "<" has been replaced by the function Equivalent_Keys. It is this function that defines the equivalence relationship for hashed maps; it is important that Equivalent_Keys(X, Y) is always the same as Equivalent_Keys(Y, X). Moreover if X and Y are equivalent and Y and Z are equivalent then X and Z must also be equivalent.

Note that the function Equivalent_Keys in the ordered maps package discussed above corresponds to the formal generic parameter of the same name in this hashed maps package. This should make it easier to convert between the two forms of packages.

Returning to our example, if we now write

```
function Equivalent_Parts(Left, Right: Part_Key) return Boolean is
begin
   return Left.Part_Number = Right.Part_Number;
end Equivalent_Parts;
```

then we can instantiate the hashed maps package as follows

```
package Store_Maps is
    new Hashed_Maps(Key_Type => Part_Key,
                    Element_Type => Stock_Info,
                    Hash => Part_Hash,
                    Equivalent_Keys => Equivalent_Parts);

The_Store: Store_Maps.Map;
```

and then the rest of our example will be exactly as before. It is thus easy to convert from an ordered map to a hashed map and vice versa provided of course that we only use the facilities common to both.

We will finish this discussion of maps by briefly considering the additional facilities in the two packages.

The ordered maps package has the following additional subprograms

```
procedure Delete_First(Container: in out Map);
procedure Delete_Last(Container: in out Map);

function First_Element(Container: Map) return Element_Type;
function First_Key(Container: Map) return Key_Type;
function Last_Element(Container: Map) return Element_Type;
function Last_Key(Container: Map) return Key_Type;
function Previous(Position: Cursor) return Cursor;
procedure Previous(Position: in out Cursor);

function Floor(Container: Map; Key: Key_Type) return Cursor;
function Ceiling(Container: Map; Key: Key_Type) return Cursor;

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

procedure Reverse_Iterate(Container: in Map;
            Process: not null access procedure (Position: in Cursor));
```

These are again largely self-evident. The functions Floor and Ceiling are interesting. Floor searches for the last node whose key is not greater than Key and similarly Ceiling searches for the first node whose key is not less than Key – they return No_Element if there is no such element. The subprograms Previous are of course the opposite of Next and Reverse_Iterate is like Iterate only backwards.

The functions "<" and ">" are mostly for convenience. Thus the first is equivalent to

```
function "<" (Left, Right: Cursor) return Boolean is
begin
  return Key(Left) < Key(Right);
end "<";
```

Clearly these additional operations must be avoided if we wish to retain the option of converting to a hashed map later.

Hashed maps have a very important facility not in ordered maps which is the ability to specify a capacity as for the vectors package. (Underneath their skin the hashed maps are a bit like vectors whereas the ordered maps are a bit like lists.) Thus we have

```
procedure Reserve_Capacity(Container: in out Map; Capacity: in Count_Type);

function Capacity(Container: Map) return Count_Type;
```

The behaviour is much as for vectors. We don't have to set the capacity ourselves since it will be automatically extended as necessary but it might significantly improve performance to do so. In the case of maps, increasing the capacity requires the hashing to be redone which could be quite time consuming, so if we know that our map is going to be a big one, it is a good idea to set an appropriate capacity right from the beginning. Note again that Length(M) cannot exceed Capacity(M) but might be much less.

The other additional subprograms for hashed maps are

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

These (like the additional "<" and ">" for ordered maps) are again mostly for convenience. The first is equivalent to

```
function Equivalent_Keys(Left, Right: Cursor) return Boolean is
begin
  return Equivalent_Keys(Key(Left), Key(Right));
end Equivalent_Keys;
```

Before moving on to sets it should be noticed that there are also some useful functions in the string packages. The main one is

```
with Ada.Containers;
function Ada.Strings.Hash(Key: String) return Containers.Hash_Type;
pragma Pure(Ada.Strings.Hash);
```

There is a similar function Ada.Strings.Unbounded.Hash where the parameter Key has type Unbounded_String. It simply converts the parameter to the type String and then calls Ada.Strings.Hash. There is also a generic function for bounded strings which again calls the basic function Ada.Strings.Hash. For completeness the function Ada.Strings.Fixed.Hash is a renaming of Ada.Strings.Hash.

These are provided because it is often the case that the key is a string and they save the user from devising good hash functions for strings which might cause a nasty headache.

We could for example save ourselves the worry of defining a good hash function in the above example by making the part number into a 5-character string. So we might write

```
function Part_Hash(P: Part_Key) return Hash_Type is
begin
  return Ada.Strings.Hash(P.Part_Number);
end Part_Hash;
```

and if this doesn't work well then we can blame the vendor.

4 Sets

Sets, like maps, come in two forms: hashed and ordered. Sets are of course just collections of values and there is no question of a key (we can perhaps think of the value as being its own key). Thus in the case of an ordered set the values are stored in order whereas in the case of a map, it is the keys that are stored in order. As well as the usual operations of inserting elements into a set and searching and so on, there are also many operations on sets as a whole that do not apply to the other containers – these are the familiar set operations such as union and intersection.

Here is the specification of the ordered sets package giving just those facilities that are common to both kinds of sets.

```
generic
  type Element_Type is private;
  with function "<" (Left, Right: Element_Type) return Boolean is <>;
  with function "=" (Left, Right: Element_Type) return Boolean is <>;
package Ada.Containers.Ordered_Sets is
  pragma Preelaborate(Ordered_Sets);

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

  type Set is tagged private;
  pragma Preelaborable_Initialization(Set);
  type Cursor is private;
  pragma Preelaborable_Initialization(Cursor);
  Empty_Set: constant Set;
  No_Element: constant Cursor;
```

The only differences from the maps package (apart from the identifiers) are that there is no key type and both "<" and "=" apply to the element type (whereas in the case of maps, the operation "<" applies to the key type). Thus the ordering relationship "<" defined on elements defines equivalence between the elements whereas "=" defines equality.

It is possible for two elements to be equivalent but not equal. For example if they were strings then we might decide that the ordering (and thus equivalence) ignored the case of letters but that equality should take the case into account. (They could also be equal but not equivalent but that is perhaps less likely.)

And as in the case of the maps package, the equality operation on elements is only used by the function "=" for comparing two sets.

Again we have the usual rules as explained for maps. Thus if x < y is true then y < x must be false; x < y and y < z must imply x < z; and x = y and y = x must be the same.

For the convenience of the user the function Equivalent_Elements is declared explicitly. It is equivalent to

```
function Equivalent_Elements(Left, Right: Element_Type) return Boolean is
begin
  return not (Left < Right) and not (Right < Left);
end Equivalent_Elements;
```

This function Equivalent_Elements corresponds to the formal generic parameter of the same name in the hashed sets package discussed below. This should make it easier to convert between the two forms of packages.

```
function "=" (Left, Right: Set) return Boolean;
function Equivalent_Sets(Left, Right: Set) return Boolean;
function To_Set(New_Item: Element_Type) return Set;
function Length(Container: Set) return Count_Type;
function Is_Empty(Container: Set) return Boolean;
procedure Clear(Container: in out Set);
```

Note the addition of Equivalent_Sets and To_Set. Two sets are equivalent if they have the same number of elements and the pairs of elements are equivalent. This contrasts with the function "=" where the pairs of elements have to be equal rather than equivalent. Remember that elements might be equivalent but not equal (as in the example of a string mentioned above). The function To_Set takes a single element and creates a set. It is particularly convenient when used in conjunction with operations such as Union described below. The other subprograms are as in the other containers.

```
function Element(Position: Cursor) return Element_Type;

procedure Replace_Element(Container: in out Set;
                          Position: in Cursor;
                          New_Item: in Element_Type);

procedure Query_Element(Position: in Cursor;
            Process: not null access procedure (Element: in Element_Type));
```

Again these are much as expected except that there is no procedure Update_Element. This is because the elements are arranged in terms of their own value (either by order or through the hash function) and if we just change an element *in situ* then it might become out of place (this problem does not arise with the other containers). This also means that Replace_Element has to ensure that the value New_Item is not equivalent to an element in a different position; if it is then Program_Error is raised. We will return to the problem of the missing Update_Element later.

```
procedure Move(Target, Source: in out Set);
```

This is just as for the other containers.

```
procedure Insert(Container: in out Set;
                    New_Item: in Element_Type;
                    Position: out Cursor;
                    Inserted: out Boolean);

procedure Insert(Container: in out Set;
                    New_Item: in Element_Type);
```

These insert a new element into the set unless an equivalent element already exists. If it does exist then the first one returns with Inserted set to False and Position indicating the element whereas the second raises Constraint_Error (the element value is not changed). If an equivalent element is not in the set then it is added and Position is set accordingly.

```
procedure Include(Container: in out Set; New_Item: in Element_Type);
```

This is somewhat like the last Insert except that if an equivalent element is already in the set then it is replaced (rather than raising Constraint_Error).

```
procedure Replace(Container: in out Set; New_Item: in Element_Type);
```

In this case, Constraint_Error is raised if an equivalent element does not already exist.

```
procedure Exclude(Container: in out Set; Item: in Element_Type);
```

If an element equivalent to Item is already in the set, then it is deleted.

```
procedure Delete(Container: in out Set; Item: in Element_Type);

procedure Delete(Container: in out Set; Position: in out Cursor);
```

These delete an element. In the first case if there is no such equivalent element then Constraint_Error is raised. In the second case if the cursor is No_Element then again Constraint_Error is also raised – there is also a check to ensure that the cursor otherwise does designate an element in the correct set (remember that cursors designate both an entity and the container); if this check fails then Program_Error is raised.

And now some new stuff, the usual set operations.

```
procedure Union(Target: in out Set; Source: in Set);
function Union(Left, Right: Set) return Set;
function "or" (Left, Right: Set) return Set renames Union;

procedure Intersection(Target: in out Set; Source: in Set);
function Intersection(Left, Right: Set) return Set;
function "and" (Left, Right: Set) return Set renames Intersection;

procedure Difference(Target: in out Set; Source: in Set);
function Difference(Left, Right: Set) return Set;
function "–" (Left, Right: Set) return Set renames Difference;

procedure Symmetric_Difference(Target: in out Set; Source: in Set);
function Symmetric_Difference (Left, Right: Set) return Set;
function "xor" (Left, Right: Set) return Set renames Symmetric_Difference;
```

These all do exactly what one would expect using the equivalence relation on the elements.

```
function Overlap(Left, Right: Set) return Boolean;
function Is_Subset(Subset: Set; Of_Set: Set) return Boolean;
```

These are self-evident as well.

```
function First(Container: Set) return Cursor;
function Last(Container: Set) return Cursor;
function Next(Position: Cursor) return Cursor;
procedure Next(Position: in out Cursor);
function Find(Container: Set; Item: Element_Type) return Cursor;
function Contains(Container: Set; Item: Element_Type) return Boolean;
```

These should be self-evident and are very similar to the corresponding operations on maps. Again unlike the operations on vectors and lists, Find logically searches the whole set and not just starting at some point (there is also no Reverse_Find). Moreover, Find uses the equivalence relation based on the "<" parameter.

```
function Has_Element(Position: Cursor) return Boolean;

procedure Iterate(Container: in Set;
            Process: not null access procedure (Position: in Cursor));
```

These are also as for other containers.

The sets packages conclude with an internal generic package called Generic_Keys. This package enables some set operations to be performed in terms of keys where the key is a function of the element. Note carefully that in the case of a map, the element is defined in terms of the key whereas here the situation is reversed. An equivalence relationship is defined for these keys as well; this is defined by a generic parameter "<" for ordered sets and Equivalent_Keys for hashed sets.

In the case of ordered sets the formal parameters are

```
generic
  type Key_Type(<>) is private;
  with function Key(Element: Element_Type) return Key_Type;
  with function "<" (Left, Right: Key_Type) return Boolean is <>;
package Generic_Keys is
```

The following are then common to the package Generic_Keys for both hashed and ordered sets.

```
function Key(Position: Cursor) return Key_Type;
function Element(Container: Set; Key: Key_Type) return Element_Type;

procedure Replace(Container: in out Set;
            Key: in Key_Type; New_Item: in Element_Type);

procedure Exclude(Container: in out Set; Key: in Key_Type);
procedure Delete(Container: in out Set; Key: in Key_Type);

function Find(Container: Set; Key: Key_Type) return Cursor;
function Contains(Container: Set; Key: Key_Type) return Boolean;

procedure Update_Element_Preserving_Key(
            Container: in out Set; Position: in Cursor;
            Process: not null access procedure (Element: in out Element_Type));
```

and then finally

```
end Generic_Keys;

private
  ... -- not specified by the language
end Ada.Containers.Ordered_Sets;
```

It is expected that most user of sets will use them in a straightforward manner and that the operations specific to sets such as Union and Intersection will be dominant.

However, sets can be used as sort of economy class maps by using the inner package Generic_Keys. Although this is certainly not for the novice we will illustrate how this might be done by reconsidering the stock problem using sets rather than maps. We declare

```
type Part_Type is
  record
    Part_Number: Integer;
    Year: Integer;
    Shelf: Character range 'A' .. 'T';
    Stock: Integer;
  end record;
```

Here we have put all the information in the one type.

We then declare "<" much as before

```
function "<" (Left, Right: Part_Type) return Boolean is
begin
  return Left.Part_Number < Right.Part_Number;
end "<";
```

and then instantiate the package thus

```
package Store_Sets is new Ordered_Sets(Element_Type => Part_Type);

The_Store: Store_Sets.Set;
```

We have used the default generic parameter mechanism for "<" this time by way of illustration.

In this case we add items to the store by calling

```
The_Store.Insert((34618, 1998, 'F', 25));
The_Store.Insert((27134, 2004, 'C', 45));
...
```

The procedure for checking the stock could now become

```
procedure Request(Part: in Integer: OK: out Boolean;
            Year: out Integer; Shelf: out Character) is
  C: Cursor;
  E: Part_Type;
begin
  C := The_Store.Find((Part, others => <>));
  if C = No_Element then
    OK := False; return;                -- no such item
  end if;
  E := Element(C);
  Year := E.Year;
  Shelf := E.Shelf;
  if E.Stock = 0 then
    OK := False; return;                -- out of stock
  end if;
  Replace_Element(C, (E.Part_Number, Year; Shelf, E.Stock–1));
  OK := True;
end Request;
```

This works but is somewhat unsatisfactory. For one thing we have had to make up dummy components in the call of Find (using <>) and moreover we have had to replace the whole of the element although we only wanted to update the Stock component. Moreover, we cannot use Update_Element because it is not defined for sets at all. Remember that this is because it might make things out of order; that wouldn't be a problem in this case because we don't want to change the part number and our ordering is just by the part number.

A better approach is to use the part number as a key. We define

```
type Part_Key is new Integer;

function Part_No(P: Part_Type) return Part_Key is
begin
  return Part_Key(P.Part_Number);
end Part_No;
```

and then

```
package Party is new Generic_Keys(Key_Type => Part_Key, Key => Part_No);
use Party;
```

Note that we do not have to define "<" on the type Part_Key at all because it already exists since Part_Key is an integer type. And the instantiation uses it by default.

And now we can rewrite the Request procedure as follows

```
procedure Request(Part: in Part_Key; OK: out Boolean;
            Year: out Integer; Shelf: out Character) is
  C: Cursor;
  E: Part_Type;
begin
  C := Find(The_Store, Part);
  if C = No_Element then
    OK := False; return;                    -- no such item
  end if;
  E := Element(C);
  Year := E.Year;  Shelf := E.Shelf;
  if E.Stock = 0 then
    OK := False; return;                    -- out of stock
  end if;

  -- we are now going to update the stock level
  declare
    procedure Do_It(E: in out Part_Type) is
    begin
      E.Stock := E.Stock – 1;
    end Do_It;
  begin
    Update_Element_Preserving_Key(The_Store, C, Do_It'Access);
  end;
  OK := True;
end Request;
```

This seems hard work but has a number of advantages. The first is that the call of Find is more natural and only involves the part number (the key) – note that this is a call of the function Find in the instantiation of Generic_Keys and takes just the part number. And the other is that the update only involves the component being changed. We mentioned earlier that there was no

Update_Element for sets because of the danger of creating a value that was in the wrong place. In the case of the richly named Update_Element_Preserving_Key it also checks to ensure that the element is indeed still in the correct place (by checking that the key is still the same); if it isn't it removes the element and raises Program_Error.

But the user is warned to take care when using the package Generic_Keys. It is absolutely vital that the relational operation and the function (Part_No) used to instantiate Generic_Keys are compatible with the ordering used to instantiate the parent package Containers.Ordered_Sets itself. If this is not the case then the sky might fall in.

Incidentally, the procedure for checking the stock which previously used the maps package now becomes

```
procedure Check_Stock(Low: in Integer) is

  procedure Check_It(C: in Cursor) is
  begin
    if Element(C).Stock < Low then
      -- print a message perhaps
      Put("Low stock of part ");
      Put_Line(Element(C).Part_Number);             -- changed
    end if;
  end Check_It;

begin
  The_Store.Iterate(Check_It'Access);
end Check_Stock;
```

The only change is that the call of Key in

```
      Put_Line(Key(C).Part_Number);
```

when using the maps package has been replaced by Element. A minor point is that we could avoid calling Element twice by declaring a constant E in Check_It thus

```
E: constant Part_Type := Element(C);
```

and then writing E.Stock < Low and calling Put_Line with E.Part_Number.

A more important point is that if we have instantiated the Generic_Keys inner package as illustrated above then we can leave Check_It unchanged to call Key. But it is important to realise that we are then calling the function Key internal to the instantiation of Generic_Keys (flippantly called Party) and not that from the instantiation of the parent ordered sets package (Store_Sets) because that has no such function. This illustrates the close affinity between the sets and maps packages.

And finally there is a hashed sets package which has strong similarities to both the ordered sets package and the hashed maps package. We can introduce this much as for hashed maps by giving the differences between the two sets packages, the extra facilities in each and the impact on the part number example.

The specification of the hashed sets package starts

```
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 <>;
package Ada.Containers.Hashed_Sets is
  pragma Preelaborate(Hashed_Sets);
```

The differences from the ordered sets package are that there is an extra generic parameter Hash and the ordering parameter "<" has been replaced by the function Equivalent_Elements.

So if we have

```
function Equivalent_Parts(Left, Right: Part_Type) return Boolean is
begin
  return Left.Part_Number = Right.Part_Number;
end Equivalent_Parts;

function Part_Hash(P: Part_Type) return Hash_Type is
  M31: constant := 2**31–1;                    -- a nice Mersenne prime
begin
  return Hash_Type(P.Part_Number) * M31;
end Part_Hash;
```

(which are very similar to the hashed map example – the only changes are to the parameter type name) then we can instantiate the hashed sets package as follows

```
package Store_Sets is
   new Hashed_Sets(Element_Type => Part_Type,
                  Hash => Part_Hash,
                  Equivalent_Elements => Equivalent_Parts);

The_Store: Store_Sets.Set;
```

and then the rest of our example will be exactly as before. It is thus easy to convert from an ordered set to a hashed set and vice versa provided of course that we only use the facilities common to both.

It should also be mentioned that the inner package Generic_Keys for hashed sets has the following formal parameters

```
generic
  type Key_Type(<>) is private;
  with function Key(Element: Element_Type) return Key_Type
  with function Hash(Key: Key_Type) return Hash_Type;
  with function Equivalent_Keys(Left, Right: Key_Type) return Boolean;
package Generic_Keys is
```

The differences from that for ordered sets are the addition of the function Hash and the replacement of the comparison operator "<" by Equivalent_Keys.

(Incidentally the package Generic_Keys for ordered sets also exports a function Equivalent_Keys for uniformity with the hashed sets package.)

Although our example itself is unchanged we do have to change the instantiation of Generic_Keys thus

```
type Part_Key is new Integer;

function Part_No(P: Part_Type) return Part_Key is
begin
  return Part_Key(P.Part_Number);
end Part_No;

function Part_Hash(P: Part_Key) return Hash_Type is
  M31: constant := 2**31–1;                    -- a nice Mersenne prime
begin
```

```
   return Hash_Type(P) * M31;
end Part_Hash;

function Equivalent_Parts(Left: Right: Part_Key) return Boolean is
begin
   return Left = Right;
end Equivalent_Parts;
```

and then

```
package Party is
   new Generic_Key(Key_Type => Part_Key,
                   Key => Part_No;
                   Hash => Part_Hash
                   Equivalent_Keys => Equivalent_Parts);
use Party;
```

The hash function is similar to that used with hashed maps. The type Part_Key and function Part_No are the same as for ordered sets. We don't really need to declare the function Equivalent_Parts since we could use "=" as the actual parameter for Equivalent_Keys.

We will finish this discussion of sets by briefly considering the additional facilities in the two sets packages (and their inner generic keys packages) just as we did for the two maps packages (the discussion is almost identical).

The ordered sets package has the following additional subprograms

```
procedure Delete_First(Container: in out Set);
procedure Delete_Last(Container: in out Set);

function First_Element(Container: Set) return Element_Type;
function Last_Element(Container: Set) return Element_Type;
function Previous(Position: Cursor) return Cursor;
procedure Previous(Position: in out Cursor);

function Floor(Container: Set; Item: Element_Type) return Cursor;
function Ceiling(Container: Set; Item: Element_Type) return Cursor;

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

procedure Reverse_Iterate(Container: in Set;
            Process: not null access procedure (Position: in Cursor));
```

These are again largely self-evident. The functions Floor and Ceiling are similar to those for ordered maps – Floor searches for the last element which is not greater than Item and Ceiling searches for the first element which is not less than Item – they return No_Element if there is not one.

The functions "<" and ">" are very important for ordered sets. The first is equivalent to

```
function "<" (Left, Right: Cursor) return Boolean is
begin
   return Element(Left) < Element(Right);
end "<";
```

There is a general philosophy that the container packages should work efficiently even if the elements themselves are very large – perhaps even other containers. We should therefore avoid copying elements. (Passing them as parameters is of course no problem since they will be passed by reference if they are large structures.) So in this case the built-in comparison is valuable because it can avoid the copying which would occur if we wrote the function ourselves with the explicit internal calls of the function Element.

On the other hand, there is a general expectation that keys will be small and so there is no corresponding problem with copying keys. Thus such built-in functions are less important for maps than sets but they are provided for maps for uniformity.

The following are additional in the package Generic_Keys for ordered sets

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

This corresponds to the formal generic parameter of the same name in the package Generic_Keys for hashed sets as mentioned earlier.

```
function Floor(Container: Set; Key: Key_Type) return Cursor;
function Ceiling(Container: Set; Key: Key_Type) return Cursor;
```

These are much as the corresponding functions in the parent package except that they use the formal parameter "<" of Generic_Keys for the search.

Hashed sets, like hashed maps also have the facility to specify a capacity as for the vectors package. Thus we have

```
procedure Reserve_Capacity(Container: in out Set; Capacity: in Count_Type);

function Capacity(Container: Set) return Count_Type;
```

The behaviour is much as for vectors and hashed maps. We don't have to set the capacity ourselves since it will be automatically extended as necessary but it might significantly improve performance to do so. Note again that Length(S) cannot exceed Capacity(S) but might be much less.

The other additional subprograms for hashed sets are

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

Again, these are very important for sets. The first is equivalent to

```
function Equivalent_Elements(Left, Right: Cursor) return Boolean is
begin
  return Equivalent_Elements(Element(Left), Element(Right));
end Equivalent_Elements;
```

and once more we see that the built-in functions can avoid the copying of the type Element that would occur if we wrote the functions ourselves.

5 Indefinite containers

There are versions of the six container packages we have just been discussing for indefinite types.

As mentioned in Section 1, an indefinite (sub)type is one for which we cannot declare an object without giving a constraint (either explicitly or though an initial value). Moreover we cannot have an array of an indefinite subtype. The type String is a good example. Thus we cannot declare an array of the type String because the components might not all be the same size and indexing would be a pain. Class wide types are also indefinite.

The specification of the indefinite container for lists starts

```
generic
   type Element_Type(<>) is private;
   with function "=" (Left, Right: Element_Type) return Boolean is <>;
package Ada.Containers.Indefinite_Doubly_Linked_Lists is
   pragma Preelaborate(Indefinite_Doubly_Linked_Lists);
```

where we see that the formal type Element_Type has unknown discriminants and so permits the actual type to be any indefinite type (and indeed a definite type as well). So if we want to manipulate lists of strings where the individual strings can be of any length then we declare

```
package String_Lists is new  Ada.Containers.Indefinite_Doubly_Linked_Lists(String);
```

In the case of ordered maps we have

```
generic
   type Key_Type(<>) is private;
   type Element_Type(<>)  is private;
   with function "<" (Left, Right: Key_Type) return Boolean is <>;
   with function "=" (Left, Right: Element_Type) return Boolean is <>;
package Ada.Containers.Indefinite_Ordered_Maps is
   pragma Preelaborate(Indefinite_Ordered_Maps);
```

showing that both Element_Type and Key_Type can be indefinite.

There are two other differences from the definite versions which should be noted.

One is that the Insert procedures for Vectors, Lists and Maps which insert an element with its default value are omitted (because there is no way to create a default initialized object of an indefinite type anyway).

The other is that the parameter Element of the access procedure Process of Update_Element (or the garrulous Update_Element_Preserving_Key in the case of sets) can be constrained even if the type Element_Type is unconstrained.

As an example of the use of an indefinite container consider the problem of creating an index. For each word in a text file we need a list of its occurrences. The individual words can be represented as just objects of the type String. It is perhaps convenient to consider strings to be the same irrespective of the case of characters and so we define

```
function Same_Strings(S, T: String) return Boolean is
begin
   return To_Lower(S) = To_Lower(T);
end Same_Strings;
```

where the function To_Lower is from the package Ada.Characters.Handling.

We can suppose that the positions of the words are described by a type Place thus

```
type Place is
   record
      Page: Text_IO.Positive_Count;
      Line: Text_IO.Positive_Count;
      Col: Text_IO.Positive_Count;
   end record;
```

The index is essentially a map from the type String to a list of values of type Place. We first create a definite list container for handling the lists thus

```
    package Places is new Doubly_Linked_Lists(Place);
```

We then create an indefinite map container from the type String to the type List thus

```
    package Indexes is new Indefinite_Hashed_Maps(
                Key_Type => String;
                Element_Type => Places.List;
                Hash => Ada.Strings.Hash;
                Equivalent_Keys => Same_Strings;
                "=" => Places."=");
```

The index is then declared by writing

```
    The_Index: Indexes.Map;
```

Note that this example illustrates the use of nested containers since the elements in the map are themselves containers (lists).

It might be helpful for the index to contain information saying which file it refers to. We can extend the type Map thus (remember that container types are tagged)

```
    type Text_Map is new Indexes.Map with
      record
        File_Ref: Text_IO.File_Access;
      end record;
```

and now we can more usefully declare

```
    My_Index: Text_Map := (Indexes.Empty_Map with My_File'Access);
```

We can now declare various subprograms to manipulate our map. For example to add a new item we have first to see whether the word is already in the index – if it is not then we add the new word to the map and set its list to a single element whereas if it is already in the index then we add the new place entry to the corresponding list. Thus

```
    procedure Add_Entry(Index: in out Text_Map; Word: String; P: Place) is
      M_Cursor: Indexes.Cursor;
      A_LIst: Places.List;                      -- empty list of places
    begin
      M_Cursor := Index.Find(Word);
      if M_Cursor = Indexes.No_Element then
        -- it's a new word
        A_LIst.Append(P);
        Index.Insert(Word, A_List);
      else
        -- it's an old word
        A_LIst := Element(M_Cursor);            -- get old list
        A_List.Append(P);                       -- add to it
        Index.Replace_Element(M_Cursor, A_LIst);
      end if;
    end Add_Entry;
```

A number of points should be observed. The type Text_Map being derived from Indexes.Map inherits all the map operations and so we can write Index.Find(Word) which uses the prefixed notation (or we can write Indexes.Find(Index, Word)). On the other hand auxiliary entities such as the type Cursor and the constant No_Element are of course in the package Indexes and have to be referred to as Indexes.Cursor and so on.

A big problem with the procedure as written however is that it uses Element and Replace_Element rather than Update_Element. This means that it copies the whole of the existing list, adds the new item to it, and then copies it back. Here is an alternative version

```
procedure Add_Entry(Index: in out Text_Map; Word: String; P: Place) is
  M_Cursor: Indexes.Cursor;
  A_LIst: Places.List;                          -- empty list of places
begin
  M_Cursor := Index.Find(Word);
  if M_Cursor = Indexes.No_Element then
    -- it's a new word
    A_LIst.Append(P);
    Index.Insert(Word, A_List);
  else
    -- it's an old word
    declare
      -- this procedure adds to the list in situ
      procedure Add_It(The_Key: in String; The_List: in out Places.List) is
      begin
        The_List.Append(P);
      end Add_It;
    begin
      -- and here we call it via Update_Element
      Index.Update_Element(M_Cursor, Add_It'Access);
    end;
  end if;
end Add_Entry;
```

This is still somewhat untidy. In the case of a new word we might as well make the new map entry with an empty list and then update it thereby sharing the calls of Append. We get

```
procedure Add_Entry(Index: in out Text_Map; Word: String; P: Place) is
  M_Cursor: Indexes.Cursor := Index.Find(Word);
  OK: Boolean;
begin
  if M_Cursor = Indexes.No_Element then
    -- it's a new word
    Index.Insert(Word, Places.Empty_List, M_Cursor, OK);
    -- M_Cursor now refers to new position
    -- and OK will be True
  end if;
  declare
   -- this procedure adds to the list in situ
    procedure Add_It(The_Key: in String; The_List: in out Places.List) is
    begin
      The_List.Append(P);
    end Add_It;
  begin
    -- and here we call it via Update_Element
    Index.Update_Element(M_Cursor, Add_It'Access);
  end;
end Add_Entry;
```

It will be recalled that there are various versions of Insert. We have used that which has two out parameters being the position where the node was inserted and a Boolean parameter indicating whether a new node was inserted or not. In this case we know that it will be inserted and so the final parameter is a nuisance (but sadly we cannot default out parameters). Note also that we need not give the parameter Places.Empty_List because another version of Insert will do that automatically since that is the default value of a list anyway.

Yet another approach is not to use Find but just call Insert. We can even use the defaulted version – if the word is present then the node is not changed and the position parameter indicates where it is, if the word is not present then a new node is made with an empty list and again the position parameter indicates where it is.

```
procedure Add_Entry(Index: in out Text_Map; Word: String; P: Place) is
   M_Cursor: Indexes.Cursor;
   Inserted: Boolean;
begin
   Index.Insert(Word, M_Cursor, Inserted);
   -- M_Cursor now refers to position of node
   -- and Inserted indicates whether it was added
   declare
     -- this procedure adds to the list in situ
      procedure Add_It(The_Key: in String; The_List: in out Places.List) is
      begin
         The_List.Append(P);
      end Add_It;
   begin
      -- and here we call it via Update_Element
      Index.Update_Element(M_Cursor, Add_It'Access);
   end;
end Add_Entry;
```

Curiously enough we do not need to use the value of Inserted. We leave the reader to decide which of the various approaches is best.

We can now do some queries on the index. For example we might want to know how many different four-lettered words there are in the text. We can either use Iterate or do it ourselves with Next as follows

```
function Four_Letters(Index: Text_Map) return Integer is
   Count: Integer := 0;
   C: Indexes.Cursor := Index.First;
begin
   loop
      if Key(C)'Length = 4 then
         Count := Count + 1;
      end if;
      Indexes.Next(C);
      exit when C = Indexes.No_Element;
   end loop;
   return Count;
end Four_Letters;
```

We might finally wish to know how many four-lettered words there are on a particular page. (This is just an exercise – it would clearly be simplest to search the original text!) We use Iterate this time both to scan the map for the words and then to scan each list for the page number

```
function Four_Letters_On_Page(Index: Text_Map;
                              Page: Text_IO.Positive_Count) return Integer is
   Count: Integer := 0;

   procedure Do_It_Map(C: Indexes.Cursor) is

      procedure Do_It_List(C: Places.Cursor) is
      begin
         if Element(C).Page = Page then
            Count := Count + 1;
         end if;
      end Do_It_LIst;

      procedure Action(K: String; E: Places.List) is
      begin
         if K'Length = 4 then
            -- now scan list for instances of Page
            E.Iterate(Do_It_List'Access);
         end if;
      end Action;

   begin
      Indexes.Query_Element(C, Action'Access);
   end Do_It_Map;

begin
   Index.Iterate(Do_It_Map'Access);
   return Count;
end Four_Letters_On_Page;
```

We could of course have used First and Next to search the list. But in any event the important point is that by using Query_Element we do not have to copy the list in order to scan it.

6 Sorting

The final facilities in the container library are generic procedures for array sorting. There are two versions, one for unconstrained arrays and one for constrained arrays. Their specifications are

```
generic
   type Index_Type is (<>);
   type Element_Type is private;
   type Array_Type is array (Index_Type range <>) of Element_Type;
   with function "<" (Left, Right: Element_Type) return Boolean is <>;
procedure Ada.Containers.Generic_Array_Sort(Container: in out Array_Type);
pragma Pure(Ada.Containers.Generic_Array_Sort);
```

and

```
generic
   type Index_Type is (<>);
   type Element_Type is private;
   type Array_Type is array (Index_Type) of Element_Type;
   with function "<" (Left, Right: Element_Type) return Boolean is <>;
```

```
procedure Ada.Containers.Generic_Constrained_Array_Sort(Container: in out Array_Type);
pragma Pure(Ada.Containers.Generic_Constrained_Array_Sort);
```

These do the obvious thing. They sort the array Container into order as defined by the generic parameter "<". The emphasis is on speed.

7 Summary table

This paper concludes with an appendix showing at a glance the various facilities in the six main containers.

References

[1] D. E. Knuth (1973). The Art of Computer Programming, vol 3 – Searching and Sorting, Addison-Wesley.

==

Appendix Container summary

In order to save space the following abbreviations are used in the table:

T	container type eg Map	H_T	Hash_Type
C: T	Container: container type	I_T	Index_Type
P: C	Position: Cursor	K_T	Key_Type
L, R	Left, Right	Ex_Index	Extended_Index
C_T	Count_Type	B	Boolean
E_T	Element_Type		

also Index – means that another subprogram exists with similar parameters except that the first parameters are of type Vector and Index_Type (or Extended_Index) rather than those involving cursors.

also Key and also Element similarly apply to maps and sets respectively.

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets
generic	Y	Y	Y	Y	Y	Y
type Index_Type is range <>;	Y					
type Key_Type is private;			Y	Y		
type Element_Type is private;	Y	Y	Y	Y	Y	Y
with function Hash(...) return Hash_Type;			on Key		on Element	
with function Equivalent_...(L, R: ...) return Boolean;			on Key		on Element	
with function "<" (L, R: ...) return Boolean is <>;				on Key		on Element
with function "=" (L, R: E_T) return B is <>;	Y	Y	Y	Y	Y	Y
package Ada.Containers.... is	Vectors	Doubly_ Linked_ Lists	Hashed_ Maps	Ordered_ Maps	Hashed_ Sets	Ordered_ Sets
pragma Preelaborate(...);	Y	Y	Y	Y	Y	Y

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets
function Equivalent_...(L, R: ...) return Boolean;				on Key		on Element
subtype Extended_Index ... No_Index: constant Ex_Ind := Ex_Ind'First;	Y					

type T is tagged private;
pragma Preelaborable_Initialization(T);
	Vector	List	Map	Map	Set	Set

type Cursor is private;
pragma Preelaborable_Initialization(Cursor);
	Y	Y	Y	Y	Y	Y

Empty_T: constant T;	Vector	List	Map	Map	Set	Set
No_Element: constant Cursor;	Y	Y	Y	Y	Y	Y
function "=" (Left, Right: T) return Boolean;	Y	Y	Y	Y	Y	Y

function Equivalent_Sets(L, R: Set) return Boolean;
function To_Set(New_Item: E_T) return Set;
					Y	Y

function To_Vector(Length: C_T) return Vector;
function To_Vector(New_Item: E_T;
Length: C_T) return Vector;
	Y					

function "&" (L, R: Vector) return Vector;
function "&" (L: Vector; R: E_T) return Vector;
function "&" (L: E_T; R: Vector) return Vector;
function "&" (L, R: E_T) return Vector;
	Y					

function Capacity(C: T) return C_T;
procedure Reserve_Capacity(C: T; Capacity: C_T);
	Y		Y		Y	

function Length(C: T) return Count_Type;	Y	Y	Y	Y	Y	Y
procedure Set_Length(C: in out T; Length: in C_T);	Y					

function Is_Empty(C: T) return B;
procedure Clear(C: in out T);
	Y	Y	Y	Y	Y	Y

function To_Cursor(C: Vector; Index: Ex_Ind)
return Cursor;
function To_Index(P: C) return Ex_Ind;
	Y					

function Key(P: C) return K_T;			Y	Y		
function Element(P: C) return E_T;	Y also Index	Y	Y	Y	Y	Y

procedure Replace_Element(C: in out T; P: C;
New_Item: E_T);
	Y
	also Index
		Y	Y	Y	Y	Y

procedure Query_Element(P: C; Process: not null acc proc(...));	in Element also Index	in Element	in Key, in Element	in Key, in Element	in Element	in Element
procedure Update_Element(C: in out T; P: C; Process: not null acc proc(...));	in out Elem also Index	in out Elem	in Key, in out Elem	in Key, in out Elem		
procedure Move(Target, Source: in out T);	Y	Y	Y	Y	Y	Y

procedure Insert(C: in out Vector; Before: Ex_Ind;
New_Item: Vector);
procedure Insert(C: in out Vector; Before: Cursor;
New_Item: Vector);
procedure Insert(C: in out Vector; Before: Cursor;
New_Item: Vector; Position: out Cursor);
	Y					

procedure Insert(C: in out T; Before: C;
New_Item: E_T; Count: C_T := 1);
	Y
	also Index
		Y				

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets
procedure Insert(C: in out T; Before: C; New_Item: E_T; Position: out Cursor; Count: C_T := 1);	Y	Y				
procedure Insert(C: in out T; Before: C; Position: out Cursor; Count: C_T := 1); element has default value	Y also Index	Y				
procedure Insert(C: in out T; Key: K_T; New_Item: E_T; Position: out Cursor; Inserted: out B);			Y	Y	Y (no key)	Y (no key)
procedure Insert(C: in out T; Key: K_T; Position: out Cursor; Inserted: out B); element has default value			Y	Y		
procedure Insert(C: in out T; Key: K_T; New_Item: E_T);			Y	Y	Y (no key)	Y (no key)
procedure Prepend(C: in out Vector; New_Item: Vector);	Y					
procedure Prepend(C: in out T; New_Item: E_T; Count: C_T := 1);	Y	Y				
procedure Append(C: in out Vector; New_Item: Vector);	Y					
procedure Append(C: in out T; New_Item: E_T; Count: C_T := 1);	Y	Y				
procedure Insert_Space(C: in out V; Before: Cursor; Position: out Cursor; Count: C_T := 1);	Y also Index					
procedure Include(C: in out T; Key: Key_Type; New_Item: E_T);			Y	Y	Y (no key)	Y (no key)
procedure Replace(C: in out T; Key: Key_Type; New_Item: E_T);			Y	Y	Y (no key)	Y (no key)
procedure Exclude(C: in out T; Key: Key_Type);			Y	Y	Y (Item not key)	Y (item not key)
procedure Delete(C: in out T; P: in out C; Count: C_T := 1);	Y also Index	Y	Y (no count) also Key	Y (no count) also Key	Y (no count) also Element	Y (no count) also Element
procedure Delete_First(C: in out T; Count: C_T := 1); procedure Delete_Last(C: in out T; Count: C_T := 1);	Y	Y		Y (no count)		Y (no count)
procedure Reverse_Elements(C: in out T);	Y	Y				
procedure Swap(C: in out T; I, J: Cursor);	Y also Index	Y				
procedure Swap_Links(C: in out List; I, J: Cursor);		Y				
procedure Splice(Target: in out List; Before: Cursor; Source: in out List); procedure Splice(Target: in out List; Before: Cursor; Source: in out List; Position: in out Cursor); procedure Splice(Container: in out List; Before: Cursor; Position: in out Cursor);		Y				
procedure Union(Target: in out Set; Source: Set); function Union(L, R: Set) return Set; function "or" (L, R: Set) return Set renames Union;					Y	Y

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets

procedure Intersection(Target: in out Set; Source: Set);
function Intersection(L, R: Set) return Set;
function "and" (L, R: Set) return Set renames Intersection;
					Y	Y

procedure Difference(Target: in out Set; Source: Set);
function Difference(L, R: Set) return Set;
function "–" (L, R: Set) return Set renames Difference;
					Y	Y

procedure Symmetric_Difference(Target: in out Set; Source: Set);
function Symmetric_Difference (L, R: Set) return Set;
function "xor" (L, R: Set) return Set renames Symmetric_Difference;
					Y	Y

function Overlap(L, R: Set) return Boolean;
function Is_Subset(Subset: Set; Of_Set: Set) return B;
					Y	Y

function First_Index(C: T) return Index_Type;	Y					
function First(C: T) return Cursor;	Y	Y	Y	Y	Y	Y
function First_Element(C: T) return Element_Type;	Y	Y		Y		Y
function First_Key(C: T) return Key_Type;				Y		
function Last_Index(C: T) return Ex_Ind;	Y					
function Last(C: T) return Cursor;	Y	Y		Y		Y
function Last_Element(C: T) return Element_Type;	Y	Y		Y		Y
function Last_Key(C: T) return Key_Type;				Y		

function Next(P: C) return Cursor;
procedure Next(P: in out C);
	Y	Y	Y	Y	Y	Y

function Previous(P: C) return Cursor;
procedure Previous(P: in out C);
	Y	Y		Y		Y

function Find_Index(C: T; Item: E_T; Index: I_T := I_T'First) return Ex_Ind;	Y					
function Find(C: T; ... ; P: C := No_Element) return Cursor;	Element	Element	Key (no position)	Key (no position)	Element (no position)	Element (no position)
function Element(C: T; Key: K_T) return E_T;			Y	Y		
function Reverse_Find_Index(C: T; Item: E_T; Index: I_T := I_T'First) return Ex_Ind;	Y					
function Reverse_Find(C: T; ... ; P: C := No_Element) return Cursor;	Element	Element				

function Floor(C: T; ...) return Cursor;
function Ceiling(C: T; ...) return Cursor;
				Key: K_T		Item: E_T

function Contains(C: T; ...) return Boolean;	Element	Element	Key	Key	Element	Element
function Has_Element(P: C) return Boolean;	Y	Y	Y	Y	Y	Y

function Equivalent_... (L, R: Cursor) return Boolean;
function Equivalent_... (L: Cursor; R:...) return Boolean;
function Equivalent_... (L:...; R: Cursor) return Boolean;
			Keys		Elements	

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets

function "<" (L, R: Cursor) return Boolean;
function ">" (L, R: Cursor) return Boolean;
function "<" (L, Cursor; R: ...) return Boolean;
function ">" (L, Cursor; R: ...) return Boolean;
function "<" (L:...; R: Cursor) return Boolean;
function ">" (L:...; R: Cursor) return Boolean;
				Key		Element

procedure Iterate(C: in T; Process: not null acc proc (P: C));	Y	Y	Y	Y	Y	Y
procedure Reverse_Iterate(C: in T; Process: not null acc proc (P: C));	Y	Y		Y		Y

generic
with function "<" (Left, Right: E_T) return B is <>;
package Generic_Sorting is
function Is_Sorted(C: T) return Boolean;
procedure Sort(C: in out T);
procedure Merge(Target, Source: in out T);
end Generic_Sorting;
	Y	Y				

generic type Key_Type (<>) is private;					Y	Y
with function Key(Element: E_T) return Key_Type;					Y	Y
with function Hash(Key: K_T) return Hash_Type;					Y	
with function Equivalent_Keys (L, R: Key_Type) return Boolean;					Y	
with function "<" (L, R: Key_Type) return B is <>;						Y
package Generic_Keys is					Y	Y
function Equivalent_Keys(L, R: Key_Type) return B;						Y
function Key(P: C) return Key_Type;					Y	Y
function Element(C: T; Key: K_T) return Element_T;					Y	Y

procedure Replace(C: in out T; Key: Key_Type;
New_Item: E_T);
procedure Exclude(C: in out T; Key: Key_Type);
procedure Delete(C: in out T; Key: Key_Type);
					Y	Y

function Find(C: T; Key: K_T) return Cursor;					Y	Y

function Floor(C: T; Key: K_T) return Cursor;
function Ceiling(C: T; Key: K_T) return Cursor;
						Y

function Contains(C: T; Key: K_T) return Boolean;					Y	Y
procedure Update_Element_Preserving_Key (C: in out T; P: C; Process: not null acc proc (Element: in out E_T));					Y	Y
end Generic_Keys;					Y	Y
private ... -- *not specified by the language* **end** Ada.Containers....;	Y	Y	Y	Y	Y	Y