John Barnes

Ada 2012 Rationale

Containers

Courtesy of AdaCore

Rationale for Ada 2012: 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 improvements to the predefined container library in Ada 2012.

Keywords: rationale, Ada 2012.

1 Overview of changes

The WG9 guidance document [1] specifically says that attention should be paid to

> improving the use and functionality of the predefined containers.

The predefined containers were introduced in Ada 2005 and experience with their use revealed a number of areas where they could be improved.

The following Ada Issues cover the relevant changes and are described in detail in this paper.

1	Bounded containers and other container issues
69	Holder container
136	Multiway tree container
139	Syntactic sugar for access, containers & iterators
159	Queue containers
184	Compatibility of streaming of containers
212	Accessors and iterators for Ada.Containers
251	Problems with queue containers

These changes can be grouped as follows.

The existing containers are unbounded and generally require dynamic storage management to be performed behind the scenes. However, for high-integrity systems, such dynamic management is often unacceptable. Accordingly, bounded versions of all the existing containers are added (1).

A number of facilities are added to make important operations on containers more elegant. These are the updating of individual elements of a container and iteration over a container (139, 212).

Ada 2005 introduced containers for the manipulation of lists and it was expected that this would provide a basis for manipulating trees. However, this proved not to be the case, so specific containers are added for the manipulation of multiway trees (136). There are versions for unbounded indefinite and unbounded definite trees and for bounded definite trees.

A further new kind of container is just for single indefinite objects and is known as the holder container (69).

A range of containers are added for manipulating queues with defined behaviour regarding multiple task access to the queues (159, 251).

The Ada 2005 container library also introduced sorting procedures for constrained and unconstrained arrays. An additional more general sorting mechanism is added in Ada 2012 (1).

Finally, an oversight regarding the streaming of containers is corrected (184).

2 Bounded and unbounded containers

It is perhaps worth starting this discussion by summarizing the containers introduced in Ada 2005. First, there is a parent package Ada.Containers which simply declares the types Hash_Type and Count_Type.

Then there are six containers for definite objects, namely (abbreviating the prefix Ada.Containers to just A.C)

```
A.C.Vectors
A.C.Doubly_Linked_Lists
A.C.Hashed_Maps
A.C.Ordered_Maps
A.C.Hashed_Sets
A.C.Ordered_Sets
```

The declarations of these six containers all start with

```
generic
  ...
  type Element_Type is private;
  ...
package Ada.Containers.XXX...
```

and we see that the type Element_Type has to be definite. There are also containers for the manipulation of indefinite types whose names are

```
A.C.Indefinite_Vectors
A.C.Indefinite_Doubly_Linked_Lists
A.C.Indefinite_Hashed_Maps
A.C.Indefinite_Ordered_Maps
A.C.Indefinite_Hashed_Sets
A.C.Indefinite_Ordered_Sets
```

and these are very similar to the definite containers except that the formal type Element_Type is now declared as

```
  type Element_Type(<>) is private;
```

so that the actual type can be indefinite such as String.

Finally, there are two generic packages for sorting arrays namely

```
A.C.Generic_Array_Sort
A.C.Generic_Constrained_Array_Sort
```

which apply to unconstrained and constrained arrays respectively.

The first change in Ada 2012 is that the parent package Ada.Containers now includes the declaration of the exception Capacity_Error so that it becomes

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

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

end Ada.Containers;
```

The names of the new containers with bounded storage capacity are

```
A.C.Bounded_Vectors
A.C.Bounded_Doubly_Linked_Lists
```

```
A.C.Bounded_Hashed_Maps
A.C.Bounded_Ordered_Maps
A.C.Bounded_Hashed_Sets
A.C.Bounded_Ordered_Sets
```

The facilities of the bounded containers are almost identical to those of the original unbounded ones so that converting a program using one form to the other is relatively straightforward. The key point of the bounded ones is that storage management is guaranteed (implementation advice really) not to use features such as pointers or dynamic allocation and therefore can be used in high-integrity or safety-critical applications.

The major differences between the packages naturally concern their capacity. In the case of the bounded packages the types such as Vector have discriminants thus

```
type Vector(Capacity: Count_Type) is tagged private;
```

whereas in the original packages the type Vector is simply

```
type Vector is tagged private;
```

The other types in the bounded packages are

```
type List(Capacity: Count_Type) is tagged private;

type Map(Capacity: Count_Type; Modulus: Hash_Type) is tagged private;

type Map(Capacity: Count_Type) is tagged private;

type Set(Capacity: Count_Type; Modulus: Hash_Type) is tagged private;

type Set(Capacity: Count_Type) is tagged private;
```

Note that the types for hashed maps and sets have an extra discriminant to set the modulus; this will be explained in a moment.

Remember that the types Count_Type and Hash_Type are declared in the parent package Ada.Containers shown above.

When a bounded container is declared, its capacity is set once and for all by the discriminant and cannot be changed. If we subsequently add more elements to the container than it can hold then the exception Capacity_Error is raised.

If we are using a bounded container and want to make it larger then we cannot. But what we can do is create another bounded container with a larger capacity and copy the values from the old container to the new one. Remember that we can check the number of items in a container by calling the function Length.

So we might have a sequence such as

```
My_List: List(100);
...                                         -- use my list
if Length(My_List) > 90 then         -- Gosh, nearly full
...
  declare
    My_Big_List: List := Copy(My_List, 200);
  begin
    ...
```

The specification of the function Copy is

```
function Copy(Source: List; Capacity: Count_Type := 0) return List;
```

If the parameter Capacity is not specified (or is given as zero) then the capacity of the copied list is the same as the length of Source.

If the given value of Capacity is larger than (or equal to) the length of the Source (as in our example) then the returned list has this capacity and the various elements are copied. If we foolishly supply a value which is less than the length of Source then Capacity_Error is naturally raised. Remember that a discriminant can be set by an initial value.

Note that if we write

```
declare
  My_Copied_List: List := My_List;
begin
```

then My_Copied_List will have the same capacity as My_List because discriminants are copied as well as the contents.

In order to make it easier to move from the bounded form to the unbounded form, a function Copy is added to the unbounded containers as well although it does not need a parameter Capacity in the case of lists and ordered maps and sets. So in the case of the list container it is simply

```
function Copy(Source: List) return List;                -- unbounded
```

Similar unification between bounded and unbounded forms occurs with assignment. In Ada 2005, if we have two lists L and M, then we can simply write

```
L := M;
```

and the whole structure is copied (including all its management stuff). Note that this will almost certainly require that the value of L be finalized which might be a nuisance. Such an assignment with discriminated types needs to check the discriminants as well (and raises Constraint_Error if they are different). This is a nuisance because although the capacities might not be the same, the destination L might have plenty of room for the actual elements in the source M.

This is all rather bothersome and so procedures Assign are added to both unbounded and bounded containers which simply copy the element values. Thus in both case we have

```
procedure Assign(Target: in out List; Source: in List);
```

In the bounded case, if the length of Source is greater than the capacity of Target, then Capacity_Error is raised. In the unbounded case, the structure is automatically extended.

It might be recalled that in Ada 2005, lists and ordered maps and sets do not explicitly have a notion of capacity. It is in their very nature that they automatically extend as required. However, in the case of vectors and hashed maps and sets (which have a notion of indexing) taking a purely automatic approach could lead to lots of extensions and copying so the notion of capacity was introduced. The capacity can be set by calling

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

and the current value of the capacity can be ascertained by calling

```
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.

The above refers to the existing unbounded forms and is unchanged in Ada 2012. For uniformity the new bounded forms for vectors and hashed maps and sets also declare a procedure Reserve_Capacity. However, since the capacity cannot be changed for the bounded forms it simply checks that the value of the parameter Capacity does not exceed the actual capacity of the container; if it does then Capacity_Error is raised and otherwise it does nothing. There is of course also a function Capacity for bounded vectors and hashed maps and sets which simply returns the fixed value of the capacity.

Many operations add elements to a container. For unbounded containers, they are automatically extended as necessary as just explained. For the bounded containers, if an operation would cause the capacity to be exceeded then Capacity_Error is raised.

There are a number of other differences between the unbounded and bounded containers. The original unbounded containers have pragma Preelaborate whereas the new bounded containers have pragma Pure.

The bounded containers for hashed maps and hashed sets are treated somewhat differently to those for the corresponding unbounded containers regarding hashing.

In the case of unbounded containers, the hashing function to be used is left to the user and is provided as an actual generic parameter. For example, in the case of hashed sets, the package specification begins

```
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);
```

What the implementation actually does with the hash function is entirely up to the implementation The value returned is in the range of Hash_Type which is a modular type declared in the root package Ada.Containers. The implementation will typically then map this value onto the current range of the capacity in some way. If the unbounded container becomes nearly full then the capacity will be automatically extended and a new mapping will be required; this in turn is likely to require the existing contents to be rehashed. None of this is visible to the user.

In the case of the new bounded containers, these problems do not arise since the capacity is fixed. Moreover, the modulus to be used for the mapping is given when the container is declared since the type has discriminants thus

```
type Set(Capacity: Count_Type; Modulus: Hash_Type) is tagged private;
```

The user can then choose the modulus explicitly or alternatively can use the additional function Default_Modulus whose specification is

```
function Default_Modulus(Capacity: Count_Type) return Hash_Type;
```

This returns an implementation defined value for the number of distinct hash values to be used for the given capacity. Thus we can write

```
My_Set: Set(Capacity => My_Cap; Modulus => Default_Modulus(My_Cap));
```

Moreover, for these bounded hashed maps and sets, the function Copy has an extra parameter thus

```
function Copy(Source: Set; Capacity: Count_Type := 0; Modulus: Hash_Type := 0)

return Set;
```

If the capacity is given as zero then the newly returned set has the same capacity as the length of Source as mentioned above. If the modulus is given as zero then the value to be used is obtained by applying Default_Modulus to the new capacity.

As mentioned in the paper on the Predefined Library, Ada 2012 introduces additional functions for hashing strings (fixed, bounded and unbounded) to provide for case insensitive, wide and wide wide situations.

Finally, note that there are no bounded containers for indefinite types. This is because the size of an object of an indefinite type (such as String) is generally not known and so indefinite types need some dynamic storage management. However, the whole point of introducing bounded containers was to avoid such management.

3 Iterating and updating containers

This topic was largely covered in the paper on Iterators and Pools which introduced the generic package Ada.Iterator.Interfaces whose specification is

```
generic
  type Cursor;
  with function Has_Element(Position: Cursor) return Boolean;
package Ada.Iterator_Interfaces is
  pragma Pure(Iterator_Interfaces);

  type Forward_Iterator is limited interface;
  function First(Object: Forward_Iterator) return Cursor is abstract;
  function Next(Object: Forward_Iterator; Position: Cursor) return Cursor is abstract;

  type Reversible_Iterator is limited interface and Forward_Iterator;
  function Last(Object: Reversible_Iterator) return Cursor is abstract;
  function Previous(Object: Reversible_Iterator; Position: Cursor) return Cursor is abstract;

end Ada.Iterator_Interfaces;
```

This generic package is used by both existing and new container packages. For illustration we consider the list container Ada.Containers.Doubly_Linked_Lists. Here is its specification giving all new and changed material in full (marked -- *12*) and identifying most existing entities by comment only.

```
with Ada.Iterator_Interfaces;                                              -- 12
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);
  pragma Remote_Types(Doubly_Linked_Lists)                                 -- 12

  type List is tagged private                                              -- 12
    with Constant_Indexing => Constant_Reference,
         Variable_Indexing => Reference,
         Default_Iterator => Iterate,
         Iterator_Element => Element_Type;
  pragma Preelaborable_Initialization(List);
  type Cursor is private;
  pragma Preelaborable_Initialization(Cursor);
  Empty_List: constant List;
  No_Element: constant Cursor;
```

```
function Has_Element(Position: Cursor) return Boolean;                    -- moved 12

package List_Iterator_Interfaces is                                             -- 12
   new Ada.Iterator_Interfaces(Cursor, Has_Element);

...        -- functions "=", Length, Is_Empty, Clear, Element
...        -- procedures Replace_, Query_, Update_Element

type Constant_Reference_Type                                                    -- 12
                (Element: not null access constant Element_Type) is private
  with Implicit_Dereference => Element;

type Reference_Type                                                             -- 12
                (Element: not null access Element_Type) is private
  with Implicit_Dereference => Element;

function Constant_Reference                                                     -- 12
                    (Container: aliased in List; Position: in Cursor)
                                        return Constant_Reference_Type;

function Reference                                                              -- 12
                    (Container: aliased in out List; Position: in Cursor)
                                              return Reference_Type;

procedure Assign(Target: in out List; Source: in List);                         -- 12

function Copy(Source: List) return List;                                        -- 12

...        -- Move, Insert, Prepend, Append,
...        -- Delete, Delete_First, Delete_Last,
...        -- Reverse_Elements, Swap, Swap_Links, Splice,
...        -- First, First_Element, Last, Last_Element,
...        -- Next, Previous, Find, Reverse_Find,
...        -- Contains, Iterate, Reverse_Iterate

function Iterate(Container: in List)                                            -- 12
                return List_Iterator_Interfaces.Reversible_Iterator'Class;

function Iterate(Container: in List; Start: in Cursor)                          -- 12
                return List_Iterator_Interfaces.Reversible_Iterator'Class;

...        -- generic package Generic_Sorting

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

Note that the function Has_Element has been moved. In Ada 2005 it was declared towards the end between Contains and Iterate. It has been moved so that it can be used as an actual parameter in the declaration of List_Iterator_Interfaces using the instantiation of Ada.Iterator_Interfaces.

It will be recalled from the paper on Iterators and Pools that in Ada 2012 we can simply write

```
for C in The_List.Iterate loop
  ...                                   -- do something via cursor C
end loop;
```

or even

```
for E of The_List loop
  ...                                -- do something to Element E
end loop;
```

rather than the laborious and error prone

```
C: The_List.Cursor;
E: Twin;
F: Forward_Iterator'Class := The_List.Iterate;
...
C := F.First;
loop
  exit when not The_List.Has_Element(C);
  E := The_List.Element(C);
  ...                                -- do something to E
  C := F.Next(C);
end loop;
```

Note that in the case of

```
for C in The_List.Iterate loop
  ...                                -- do something via cursor C
end loop;
```

we are not permitted to assign to C since that would upset the mechanism of the loop. There is an analogy with the traditional loop statement. If we write

```
for K in A'Range loop
  A(K) := 0;
end loop;
```

then the language prevents us from making a direct assignment to the loop parameter K.

If we write

```
for E of The_List loop
  ...                                -- do something to Element E
end loop;
```

then we can change the element E unless The_List has been declared as constant.

It will be recalled that subprograms Replace_Element, Query_Element and Update_Element are defined for all containers in Ada 2005. Query_Element and Update_Element permit *in situ* operations. Thus in order to find the value of some component Q of an element of The_List identified by cursor C we can write either

```
X := Element(C).Q;
```

or we can first declare a slave procedure

```
procedure Get_Q(E: in Element_Type) is
begin
  X := E.Q;
end Get_Q;
```

and then call Query_Element thus

```
Query_Element(C, Get_Q'Access);
```

The advantage of the former is that it is easy but it could be slow because it copies the whole element which could be enormous. The advantage of the latter is that it does not copy the element; its disadvantage is that it is somewhat incomprehensible.

In Ada 2012, we can do much better. The type List now has new functions Reference and Constant_Reference, so we can write for example

```
X := The_List.Constant_Reference(C).Q;
```

This works because the function Constant_Reference returns a value of Constant_Reference_Type and this moreover has aspect Implicit_Dereference whose value is Element.

However, we can simplify this even more because the type List has aspects Constant_Indexing and Variable_Indexing which refer to the functions Constant_Reference and Reference. The result is that we can simply write

```
X := The_List(C).Q;                    -- gosh that's better
```

which is a lot better than calling Query_Element.

Similarly, if we just want to update the component Q of some element given by a cursor C, then in Ada 2005 we either have to create a whole new element with the new value for Q and then use Replace_Element thus

```
Temp: E_Type := Element(C);
...
Temp.Q := X;

Replace_Element(The_List, C, Temp);
```

or declare a slave procedure and use Update_Element thus

```
procedure Put_Q(E: in out Element_Type) is
begin
  E.Q := X;
end Put_Q;

Update_Element(The_List, C, Put_Q'Access);
```

Again the first is slow, the second is gruesome (well, they are both gruesome really).

In Ada 2012 we simply write

```
The_List(C).Q := X;                    -- gosh again
```

which implicitly uses the aspect Variable_Indexing to call the function Reference which gives access to the element.

It will be remembered that there are dire warnings in Ada 2005 about tampering with elements and cursors. Thus we must not use Update_Element (that is via Put_Q in the example above) to do other things such as add new elements.

Although tampering is still possible in Ada 2012; the new features discourage it. Thus if we write

```
The_List(C).Q := X;
```

rather than calling Update_Element then no tampering can occur (unless X is some gruesome function).

Similarly if we write

```
for C in My_Container loop
  ...
```

```
      Delete(My_Container, Position => C);               --illegal
      ...
   end loop;
```

then we are prevented from madness since the parameter Position of Delete is of mode **in out** and this is not matched by the loop parameter C which is a constant. However, if we write the loop out using First and Next as illustrated earlier then we could get into trouble.

4 Multiway tree containers

Three new containers are added for multiway trees; two correspond to the existing unbounded definite and unbounded indefinite forms for existing structures such as Lists and Maps in Ada 2005. There is also a bounded form corresponding to the newly introduced bounded containers for the existing structures discussed above. As expected their names are

```
   A.C.Multiway_Trees
   A.C.Indefinite_Multiway_Trees
   A.C.Bounded_Multiway_Trees
```

These containers have all the operations required to operate on a tree structure where each node can have multiple child nodes to any depth. Thus there are operations on subtrees, the ability to find siblings, to insert and remove children and so on. It will be noted that many of the operations on trees are similar to corresponding operations on lists.

We will look in detail at the unbounded definite form by giving its specification interspersed with some explanation. It starts with the usual generic parameters.

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

      type Tree is tagged private
         with Constant_Indexing => Constant_Reference,
                 Variable_Indexing => Reference,
                 Default_Iterator => Iterate,
                 Iterator_Element => Element_Type;
      pragma Preelaborable_Initialization(Tree);
      type Cursor is private;
      pragma Preelaborable_Initialization(Cursor);
      Empty_Tree: constant Tree;
      No_Element: constant Cursor;

      function Has_Element(Position: Cursor) return Boolean;
      package Tree_Iterator_Interfaces is
         new Ada.Iterator_Interfaces(Cursor, Has_Element);
```

This is much as expected and follows the same pattern as the start of the list container in the previous section.

```
      function Equal_Subtree(Left_Position: Cursor; Right_Position: Cursor) return Boolean;
      function "=" (Left, Right: Tree) return Boolean;

      function Is_Empty(Container: Tree) return Boolean;
```

```
function Node_Count(Container: Tree) return Count_Type;
function Subtree_Node_Count(Position: Cursor) return Count_Type;

function Depth(Position: Cursor) return Count_Type;

function Is_Root(Position: Cursor) return Boolean;
function Is_Leaf(Position: Cursor) return Boolean;
function Root(Container: Tree) return Cursor;
procedure Clear(Container: in out Tree);
```

A tree consists of a set of nodes linked together in a hierarchical manner. Nodes are identified as usual by the value of a cursor. Nodes can have one or more child nodes; the children are ordered so that there is a first child and a last child. Nodes with the same parent are siblings. One node is the root of the tree. If a node has no children then it is a leaf node.

All nodes other than the root node have an associated element whose type is Element_Type. The whole purpose of the tree is of course to give access to these element values in a structured manner.

The function "=" compares two trees and returns true if and only if they have the same structure of nodes and corresponding nodes have the same values as determined by the generic parameter "=" for comparing elements. Similarly, the function Equal_Subtree compares two subtrees.

The function Node_Count gives the number of nodes in a tree. All trees have at least one node, the root node. The function Is_Empty returns true only if the tree consists of just this root node. Note that A_Tree = Empty_Tree, Node_Count(A_Tree) = 1 and Is_Empty(A_Tree) always have the same value. The function Subtree_Node_Count returns the number of nodes in the subtree identified by the cursor. If the cursor value is No_Element then the result is zero.

The functions Is_Root and Is_Leaf indicate whether a node is the root or a leaf respectively. If a tree is empty and so consists of just a root node then that node is both the root and a leaf so both functions return true.

The function Depth returns 1 if the node is the root, and otherwise indicates the number of ancestor nodes. Thus a node which is an immediate child of the root has depth equal to 2. The function Root returns the cursor designating the root of a tree. The procedure Clear removes all elements from the tree so that it consists just of a root node.

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

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

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

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

These subprograms have the expected behaviour similar to other containers.

```
type Constant Reference_Type(Element: not null access constant Element_Type)

  is private
    with Implicit_Dereference => Element;

type Reference_Type(Element: not null access Element_Type) is private
  with Implicit_Dereference => Element;
```

```
    function Constant_Reference(Container: aliased in Tree; Position: in Cursor)
                                                                    return
  Constant_Reference_Type;

    function Reference(Container: aliased in out Tree; Position: in Cursor)
                                                                             return
  Reference_Type;
```

These types and functions are similar to those for the other containers and were explained in the paper on Iterators and Pools and also in the previous section.

```
    procedure Assign(Target: in out Tree; Source: in Tree);

    function Copy(Source: Tree) return Tree;

    procedure Move(Target: in out Tree; Source: in out Tree);
```

The subprograms Assign and Copy behave as expected and were explained in the section on Bounded and Unbounded containers. The procedure Move moves all the nodes from the source to the target after first clearing the target; it does not make copies of the elements so after the operation the source only has a root node.

```
    procedure Delete_Leaf(Container: in out Tree; Position: in out Cursor);

    procedure Delete_Subtree(Container: in out Tree; Position: in out Cursor);

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

The procedures Delete_Leaf and Delete_Subtree check that the cursor value designates a node of the container and raise Program_Error if it does not. Program_Error is also raised if Position designates the root node and so cannot be removed. In the case of Delete_Leaf, if the node has any children then Constraint_Error is raised. The appropriate nodes are then deleted and Position is set to No_Element.

The procedure Swap interchanges the values in the two elements denoted by the two cursors. The elements must be in the given container (and must not denote the root) otherwise Program_Error is raised.

```
    function Find(Container: Tree; Item: Element_Type) return Cursor;

    function Find_In_Subtree(Item: Element_Type; Position: Cursor) return Cursor;

    function Ancestor_Find(Item: Element_Type; Position: Cursor) return Cursor;

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

These search for an element in the container with the given value Item. The function Contains returns false if the item is not found; the other functions return No_Element if the item is not found. The function Find searches the whole tree starting at the root node, Find_In_Subtree searches the subtree rooted at the node given by Position including the node itself; these searches are in depth-first order. The function Ancestor_Find searches upwards through the ancestors of the node given by Position including the node itself.

Depth-first order is explained at the end of the section.

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

    procedure Iterate_Subtree(Position: in Cursor;
                                      Process: not null access procedure (Position: in
  Cursor));
```

These apply the procedure designated by the parameter Process to each element of the whole tree or the subtree. This includes the node at the subtree but not at the root; iteration is in depth-first order.

```
   function Iterate(Container: in Tree) return Tree_Iterator_Interfaces.Forward_Iterator'Class;

   function Iterate_Subtree(Position: in Cursor)
                                                    return
Tree_Iterator_Interfaces.Forward_Iterator'Class;
```

The first of these is called if we write

```
for C in The_Tree.Iterate loop
   ...                          -- do something via cursor C
end loop;
```

and iterates over the whole tree in the usual depth-first order. In order to iterate over a subtree we write

```
for C in The_Tree.Iterate(S) loop
   ...                          -- do something via cursor C
end loop;
```

and this iterates over the subtree rooted at the cursor position given by S.

If we use the other new form of loop using **of** thus

```
for E of The_Tree loop
   ...                          -- do something to element E
end loop;
```

then this also calls Iterate since the aspect Default_Iterator of the type Tree (see above) is Iterate. However, we cannot iterate over a subtree using this mechanism.

```
   function Child_Count(Parent: Cursor) return Count_Type;

   function Child_Depth(Parent, Child: Cursor) return Count_Type;
```

The function Child_Count returns the number of child nodes of the node denoted by Parent. This count covers immediate children only and not grandchildren.

The function Child_Depth indicates how many ancestors there are from Child to Parent. If Child is an immediate child of Parent then the result is 1; if it is a grandchild then 2 and so on.

```
   procedure Insert_Child(Container: in out Tree;
                          Parent: in Cursor;
                          Before: in Cursor;
                          New_Item: in Element_Type;
                          Count: in Count_Type := 1);

   procedure Insert_Child(Container: in out Tree;
                          Parent: in Cursor;
                          Before: in Cursor;
                          New_Item: in Element_Type;
                          Position: out Cursor;
                          Count: in Count_Type := 1);

   procedure Insert_Child(Container: in out Tree;
                          Parent: in Cursor;
                          Before: in Cursor;
                          Position: out Cursor;
                          Count: in Count_Type := 1);
```

These three procedures enable one or more new child nodes to be inserted. The parent node is given by Parent. If Parent already has children then the new nodes are inserted before the child node identified by Before; if Before is No_Element then the new nodes are inserted after all existing children. The second procedure is similar to the first but also returns a cursor to the first of the added nodes. The third is like the second but the new elements take their default values. Note the default value of one for the number of new nodes.

```
procedure Prepend_Child(Container: in out Tree;
                        Parent: in Cursor;
                        New_Item: in Element_Type;
                        Count: in Count_Type := 1);

procedure Append_Child(Container: in out Tree;
                       Parent: in Cursor;
                       New_Item: in Element_Type;
                       Count: in Count_Type:= 1);
```

These insert the new children before or after any existing children.

```
procedure Delete_Children(Container: in out Tree;
                          Parent: in Cursor);
```

This procedure simply deletes all the children, grandchildren, and so on of the node designated by Parent.

```
procedure Copy_Subtree(Target: in out Tree;
                       Parent: in Cursor;
                       Before: in Cursor;
                       Source: in Cursor);
```

This copies the complete subtree rooted at Source into the tree denoted by Tree as a subtree of Parent at the place denoted by Before using the same rules as Insert_Child. Note that this makes a complete copy and creates new nodes with values equal to the corresponding existing nodes. Note also that Source might be within Tree but might not. There are the usual various checks.

```
procedure Splice_Subtree(Target: in out Tree;
                         Parent: in Cursor;
                         Before: in Cursor;
                         Source: in out Tree;
                         Position: in out Cursor);

procedure Splice_Subtree(Container: in out Tree;
                         Parent: in Cursor;
                         Before: in Cursor;
                         Position: in Cursor);

procedure Splice_Children(Target: in out Tree;
                          Target_Parent: in Cursor;
                          Before: in Cursor;
                          Source: in out Tree;
                          Source_Parent: in Cursor);

procedure Splice_Children(Container: in out Tree;
                          Target_Parent: in Cursor;
                          Before: in Cursor;
                          Source_Parent: in Cursor);
```

These are similar to the procedures Splice applying to lists. They enable nodes to be moved without copying. The destination is indicated by Parent or Target_Parent together with Before as usual indicating where the moved nodes are to be placed with respect to existing children of Parent or Target_Parent.

The first Splice_Subtree moves the subtree rooted at Position in the tree Source to be a child of Parent in the tree Target. Note that Position is updated to be the appropriate element of Target. We can use this procedure to move a subtree within a tree but an attempt to create circularities raises Program_Error.

The second Slice_Subtree is similar but only moves a subtree within a container. Again, circularities cannot be created.

The procedures Splice_Children are similar but move all the children and their descendants of Source_Parent to be children of Target_Parent.

```
function Parent(Position: Cursor) return Cursor;
function First_Child(Parent: Cursor) return Cursor;
function First_Child_Element(Parent: Cursor) return Element_Type;
function Last_Child(Parent: Cursor) return Cursor;
function Last_Child_Element(Parent: Cursor) return Element_Type;
function Next_Sibling(Position: Cursor) return Cursor;
function Previous_Sibling(Position: Cursor) return Cursor;
procedure Next_Sibling(Position: in out Cursor);
procedure Previous_Sibling(Position: in out Cursor);
```

Hopefully, the purpose of these is self-evident.

```
procedure Iterate_Children(Parent: in Cursor;
                           Process: not null access procedure (Position: in
Cursor));

procedure Reverse_Iterate_Children(Parent : in Cursor;
                           Process: not null access procedure (Position: in
Cursor));
```

These apply the procedure designated by the parameter Process to each child of the node given by Parent. The procedure Iterate_Children starts with the first child and ends with the last child whereas Reverse_Iterate_Children starts with the last child and ends with the first child. Note that these do not iterate over grandchildren.

```
function Iterate_Children(Container: in Tree; Parent: in Cursor) return

          Tree_Iterator_Interfaces.Reversible_Iterator'Class;
```

This is called if we write

```
for C in Parent.Iterate_Children loop
   ...                          -- do something via cursor C
end loop;
```

and iterates over all the children from Parent.First_Child to Parent.Last_Child. Note that we could also insert **reverse** thus

```
for C in reverse Parent.Iterate_Children loop
   ...                          -- do something via cursor C
end loop;
```

in which case the iteration goes in reverse from Parent.Last_Child to Parent.First_Child. The observant reader will note that this function returns Reversible_Iterator'Class and so can go in either direction whereas the functions Iterate and Iterate_Subtree described earlier use Forward_Iterator'Class and cannot be reversed.

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

The above descriptions have not described all the situations in which something can go wrong and so raise Constraint_Error or Program_Error. Generally, the former is raised if a source or target is No_Element; the latter is raised if a cursor does not belong to the appropriate tree. In particular, as mentioned above, an attempt to create an illegal tree such as one with circularities using Splice_Subtree raises Program_Error. Remember also that every tree has a root node but the root node has no element value; attempts to remove the root node or read its value or assign a value similarly raise Program_Error.

The containers for indefinite and bounded trees are much as expected.

In the case of the indefinite tree container the generic formal type is

```
type Element_Type(<>) is private;
```

The other significant difference is that the procedure Insert_Child without the parameter New_Item is omitted; this is because indefinite types do not have a default value.

In the case of the bounded tree container the changes are similar to those for the other containers. One change is that the package has pragma Pure; the other changes concern the capacity. The type Tree is

```
type Tree(Capacity: Count_Type) is tagged private;
```

and the function Copy is

```
function Copy(Source: Tree; Capacity: Count_Type := 0) return Tree;
```

And of course the exception Capacity_Error is raised in various circumstances.

Applications of trees are usually fairly complex. The tree structure for depicting the analysis of a program for a whole language such as even Ada 83 has an enormous variety of nodes corresponding to the various syntactic structures. And trees depicting human relationships are complex because of multiple marriages, divorces, illegitimacy and so on. So we content ourselves with a couple of small examples.

A tree representing a simple algebraic expression involving just the binary operations of addition, subtraction, multiplication and division applied to simple variables and real literals is straightforward. Nodes are of three kinds, those representing operations have two children giving the two operands, and those representing variables and literals have no children and so are leaf nodes.

We can declare the element type thus

```
type Operator is ('+', '–', '×', '/');
type Kind is (Op, Var, Lit);

type El(K: Kind) is
   record
      case K is
         when Op =>
            Fn: Operator;
         when Var =>
      V: Character;
    when Lit =>
      Val: Float;
  end case;
end record;
```

Note that the variables are (as typically in mathematics) represented by single letters. So the expression

$(x + 3) \times (y - 4)$

is represented by nodes with elements such as

```
(Op, '×')
(Var, 'x')
(Lit, 3.0)
```

So now we can declare a suitable tree thus

```
package Expression_Trees is
  new Ada.Containers.Multiway_Trees(El);

use Expression_Trees;

My_Tree: Tree := Empty_Tree;

C: Cursor;
```

and then build it by the following statements

```
C := Root(My_Tree);

Insert_Child(Container => My_Tree,
                      Parent => C,
                      Before => No_Element,
                      New_Item => (Op, '×'),
                      Position => C);
```

This puts in the first real node as a child of the root which is designated by the cursor C. There are no existing children so Before is No_Element. The New_Item is as mentioned earlier. Finally, the cursor C is changed to designate the position of the newly inserted node.

We can then insert the two children of this node which represent the mathematical operations + (plus) and – (minus).

```
Insert_Child(My_Tree, C, No_Element, (Op, '+'));
Insert_Child(My_Tree, C, No_Element, (Op, '–'));
```

These calls are to a different overloading of Insert_Child and have not changed the cursor. The second call also has Before equal to No_Element and so the second child goes after the first child. We now change the cursor to that of the first newly inserted child and then insert its children which represent *x* and 3. Thus

```
C := First_Child(C);
Insert_Child(My_Tree, C, No_Element, (Var, 'x'));
Insert_Child(My_Tree, C, No_Element, (Lit, 3.0));
```

And then we can complete the tree by inserting the final two nodes thus

```
C := Next_Sibling(C);
Insert_Child(My_Tree, C, No_Element, (Var, 'y'));
Insert_Child(My_Tree, C, No_Element, (Lit, 4.0));
```

Of course a compiler will do all this recursively and keep track of the cursor rather more neatly than we have in this manual illustration.

The resulting tree should be as in Figure 1.

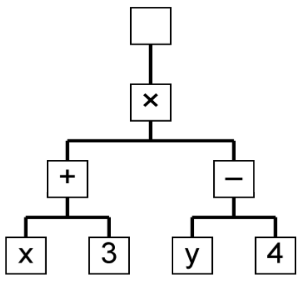

Figure 1 The expression tree

We can assume that the variables are held in an array which might be as follows

```
subtype Variable_Name is Character range 'a' .. 'z';

Variables: array (Variable_Name) of Float;
```

We can then evaluate the tree by a recursive function such as

```
function Eval(C: Cursor) return Float is
  E: El := Element(C);
  L, R: Float
begin
  case E.K is
    when Op =>
      L := Eval(First_Child(C));
      R := Eval(Last_Child(C));
      case E.Fn is
        when '+' => return (L+R);
        when '–' => return (L–R);
        when '×' => return (L*R);
        when '/' => return (L/R);
      end case;
    when Var =>
      return Variables(E.V);
    when Lit =>
      return E.Val;
  end case;
end Eval;
```

Finally, we obtain the value of the tree by

```
X := Eval(First_Child(Root(My_Tree)));
```

Remember that the node at the root has no element so hence the call of First_Child.

An alternative approach would be to use tagged types with a different type for each kind of node rather than the variant record. This would be much more flexible and would have required the use of the unbounded indefinite container Ada.Containers.Indefinite_Multiway_Trees.

As a more human example we can consider the family tree of the Tudor Kings and Queens of England. We start with Henry VII, who had four children, Arthur, Margaret, Henry VIII and Mary. See Figure 2.

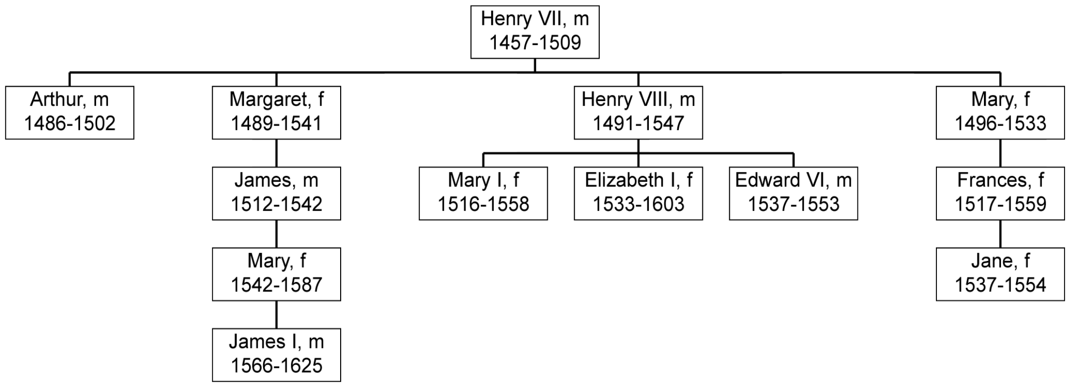

Figure 2 The Tudor tree

Arthur died young, Margaret married James IV of Scotland and had James (who was thus James V of Scotland), Henry VIII had three children, namely Edward VI, Mary I and Elizabeth I. And Mary had Frances. Henry VII was succeeded by Henry VIII and he was succeeded by his three children.

Remember the rules of primogeniture. The heir is the eldest son if there are sons; if not then the heir is the eldest daughter. If there are no offspring at all then we go back a generation and try again. Hence Edward VI became king despite being younger than Mary.

Since Edward, Mary and Elizabeth had no children we go back to the descendants of the other children of Henry VII. Margaret, her son James, and his daughter Mary Queen of Scots were all dead by then, so the throne of England went to the son of Mary who became James I of England and VI of Scotland and thus united the two thrones. So the Tudor line died with Elizabeth (Good Queen Bess).

Incidentally, Frances, the daughter of Mary, the fourth child of Henry VII, had a daughter, Lady Jane Grey; she was Queen for 9 days but lost her head over a row with Mary I.

Representing this is tricky, especially with people such as Henry VIII having so many wives. But the essence could be represented by a tree with a simple element type thus

```
type Person is
  record
    Name: String(1 .. 10);
    Sex: Gender;
    Birth: Date;
    Death: Date;
  end record;
```

With such a structure and the dates, starting from Henry VII and using the rules of primogeniture, one should be able to trace the monarchs (apart from poor Lady Jane Grey who would I am sure much rather not have been involved).

The overall tree structure is shown in Figure 2.

With the obvious connections we can define useful functions such as

```
function Are_Cousins(A, B: Cursor) return Boolean is
  (Parent(A) /= Parent (B) and then Parent(Parent(A)) = Parent(Parent(B)));
```

More of a challenge is to define a function Is_Successor using the rules described above. The reader can contemplate these and other family relationships and attempt to construct the Tudor tree.

Finally, an explanation of depth-first order. The general principle is that child nodes are visited in order before their parent. We can symbolically write this as

```
procedure Do_Node(N) is
begin
  for CN in N.First_Child .. N.Last_Child loop
    Do_Node(CN);
  end loop;
  if not N.Is_Root then
    Do_Element(N);
  end if;
end Do_Node;
```

and the whole thing is triggered by calling Do_Node(Root). Remember that the root node has no element. The result is that the first element to be processed is that of the leftmost leaf.

Thus in the tree illustrated below in Figure 3, the elements are visited in order A, B, C, D, and so on. Note that the root has no element and so is not visited.

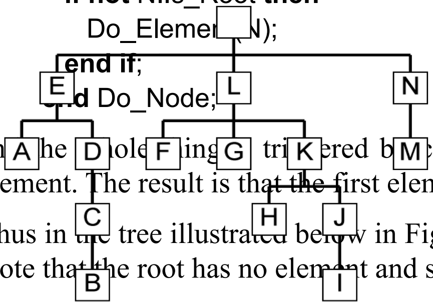

Figure 3 A tree showing depth-order first

5 The holder container

As mentioned in the Introduction, it is not possible to declare an object of an indefinite type that can hold any value of that type since the object becomes constrained by the mandatory initial value. Thus we can write

```
Pet: String := "dog";
```

We can assign "cat" to Pet but we cannot assign "rabbit" because it is too long.

This is overcome in Ada 2012 by the introduction of the holder container which can hold a single indefinite object. Its specification is

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

```
  pragma Preelaborate(Indefinite_Holders);
  pragma Remote_Types(Indefinite_Holders);

  type Holder is tagged private;
  pragma Preelaborable_Initialization(Holder);

  Empty_Holder: constant Holder;

  function "=" (Left, Right: Holder) return Boolean;

  function To_Holder(New_Item: Element_Type) return Holder;

  function Is_Empty(Container: Holder) return Boolean;

  procedure Clear(Container: in out Holder);

  function Element(Container: Holder) return Element_Type;

  procedure Replace_Element(Container: in out Holder; New_Item: in Element_Type);

  procedure Query_Element(Container: in Holder;
                          Process: not null access procedure (Element: in Element_Type));

  procedure Update_Element(Container: in out Holder;
                          Process: not null access procedure (Element: in out
Element_Type));

  type Constant_Reference_Type(Element: not null access constant Element_Type)
                                                                              is
private
    with Implicit_Dereference => Element;

  type Reference_Type(Element: not null access Element_Type) is private
    with Implicit_Dereference => Element;

  function Constant_Reference(Container: aliased in Holder)
                                                         return
Constant_Reference_Type;

  function Reference(Container: aliased in out Holder) return Reference_Type;

  procedure Assign(Target: in out Holder; Source: in Holder);

  function Copy(Source: Holder) return Holder;

  procedure Move(Target: in out Holder; Source: in out Holder);

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

Hopefully, the purpose of the facilities provided by this container are obvious given an understanding of the use of the existing containers. It would be possible to use a list container with just a single element to act as a holder but it seems better to have an explicit container with probably less overhead and risk of confusion.

A trivial example of its use might be to provide a holder for pets. We write

```
package Strings is
  new Ada.Containers.Indefinite_Holders(String);

Kennel: Strings.Holder := To_Holder("cat");
```

This declares an object Kennel which is a wrapper for a string and initializes it with the string "cat". We can replace the cat with a rabbit by writing

```
Kennel := To_Holder("rabbit");
```

However, using To_Holder in this way could be a bit slow since this will create a new object which has to be destroyed after the assignment. It is better to write

```
Replace_Element(Kennel, "rabbit");
```

If we want to print out the contents of the kennel we just write

```
Put(Element(Kennel));
```

Operations such as Update_Element are provided partly for uniformity but also because the object might be large so that it is better to update it *in situ*. Alternatively, we can use the functions such as Reference as explained earlier.

6 Queue containers

When the goals of the revision to Ada 2005 were discussed, one of the expectations was that it would be possible to improve the containers, or maybe introduce variants, that would be task safe. However, further investigation revealed that this would not be practicable because the number of ways in which several tasks could interact with a container such as a list or map was large.

However, one data structure that is amenable to controlled access by several tasks is the queue. One or more tasks can place objects on a queue and one or more can remove them. Moreover, the existing container library did not include queues as such so we were not tied to any existing structures.

There are in fact four different queue containers in Ada 2012. These are all for elements of a definite type. Two are bounded and two are unbounded. And there are priority and synchronized queues. The names are

```
A.C.Unbounded_Synchronized_Queues
A.C.Bounded_Synchronized_Queues
A.C.Unbounded_Priority_Queues
A.C.Bounded_Priority_Queues
```

At one stage it was also planned to have unbounded containers for elements of an indefinite type. This would then have been similar to the other containers which have unbounded definite, unbounded indefinite and bounded definite forms. However, there were significant problems with the Dequeue operation to remove an indefinite object related to the fact that Ada does not have entry functions. This is easily overcome by making the elements of the queue a holder container as described in the previous section.

These four different queue containers are all derived from a single synchronized interface declared in a generic package whose specification is as follows

```
generic
  type Element_Type is private;                    -- definite
package A.C.Synchronized_Queue_Interfaces is
  pragma Pure(Synchronized_Queue_Interfaces);

  type Queue is synchronized interface;

  procedure Enqueue(Container: in out Queue; New_Item: in Element_Type) is abstract
    with Synchronization => By_Entry;
```

```
   procedure Dequeue(Container: in out Queue; Element: out Element_Type) is abstract
     with Synchronization => By_Entry;

   function Current_Use(Container: Queue) return Count_Type is abstract;
   function Peak_Use(Container: Queue) return Count_Type is abstract;
end A.C.Synchronized_Queue_Interfaces;
```

This generic package declares the synchronized interface Queue and four operations on queues. These are the procedures Enqueue and Dequeue to add items to a queue and remove items from a queue respectively; note the aspect Synchronization which ensures that all implementations of these abstract procedures must be by an entry. There are also functions Current_Use and Peak_Use which can be used to monitor the number of items on a queue.

The four queue containers are generic packages which themselves declare a type Queue derived in turn from the interface Queue declared in the package above. We will look first at the synchronized queues and then at the priority queues.

The package for unbounded synchronized queues is as follows

```
with System; use System;
with A.C.Synchronized_Queue_Interfaces;
generic
   with package Queue_Interfaces is new A.C.Synchronized_Queue_Interfaces(<>);
   Default_Ceiling: Any_Priority := Priority'Last;
package A.C.Unbounded_Synchronized_Queues is
   pragma Preelaborate(Unbounded_Synchronized_Queues);

   package Implementation is
      ...               -- not specified by the language
   end Implementation;

   protected type Queue(Ceiling: Any_Priority := Default_Ceiling)
     with Priority => Ceiling
                              is new Queue_Interfaces.Queue with

     overriding
     entry Enqueue(New_Item: in Queue_Interfaces.Element_Type);
     overriding
     entry Dequeue(Element: out Queue_Interfaces.Element_Type);

     overriding
     function Current_Use return Count_Type;
     overriding
     function Peak_Use return Count_Type;

   private
     ...                -- not specified by the language
   end Queue;

private
   ...                  -- not specified by the language
end A.C.Unbounded_Synchronized_Queues;
```

Note that there are two generic parameters. The first (Queue_Interfaces) has to be an instantiation of the interface generic Synchronized_Queue_Interfaces; remember that the parameter (<>) means that any instantiation will do. The second parameter concerns priority and has a default value so we can ignore it for the moment.

Inside this package there is a protected type Queue which controls access to the queues via its entries Enqueue and Dequeue. This protected type is derived from Queue_Interfaces.Queue and so promises to implement the operations Enqueue, Dequeue, Current_Use and Peak_Use of that interface. And indeed it does implement them and moreover implements Enqueue and Dequeue by entries as required by the aspect Synchronization.

As an example suppose we wish to create a queue of some records such as

```
type Rec is record ... end record;
```

First of all we instantiate the interface package (using named notation for clarity) thus

```
package Rec_Interface is
  new A.C.Synchronized_Queue_Interfaces(Element_Type => Rec);
```

This creates an interface from which we can create various queuing mechanisms for dealing with objects of the type Rec.

Thus we might write

```
package Unbounded_Rec_Package is
  new A.C.Unbounded_Synchronized_Queues(Queue_Interfaces => Rec_Interface);
```

Finally, we can declare a protected object, My_Rec_UQ which is the actual queue, thus

```
My_Rec_UQ: Unbounded_Rec_Package.Queue;
```

To place an object on the queue we can write

```
Enqueue(My_Rec_UQ, Some_Rec);
```

or perhaps more neatly

```
My_Rec_UQ.Enqueue(Some_Rec);
```

And to remove an item from the queue we can write

```
My_Rec_UQ.Dequeue(The_Rec);
```

where The_Rec is some object of type Rec which thereby is given the value removed.

The statement

```
N := Current_Use(My_Rec_UQ);
```

assigns to N the number of items on the queue when Current_Use was called (it could be out of date by the time it gets into N) and similarly Peak_Use(My_Rec_UQ) gives the maximum number of items that have been on the queue since it was declared.

This is all task safe because of the protected type; several tasks can place items on the queue and several, perhaps the same, can remove items from the queue without interference.

It should also be noticed that since the queue is unbounded, we never get blocked by Enqueue since extra storage is allocated as required just as for the other unbounded containers (I suppose we might get Storage_Error).

The observant reader will note the mysterious local package called Implementation. This enables the implementation to declare local types to be used by the protected type. It will be recalled that there is an old rule that one cannot declare a type within a type. These local types really ought to be within the private part of the protected type; maybe this is something for Ada 2020.

The package for bounded synchronized queues is very similar. The only differences (apart from its name) are that it has an additional generic parameter Default_Capacity and the protected type Queue has an additional discriminant Capacity. So its specification is

```
with System; use System;
with A.C.Synchronized_Queue_Interfaces;
generic
  with package Queue_Interfaces is new A.C.Synchronized_Queue_Interfaces(<>);
  Default_Capacity: Count_Type;
  Default_Ceiling: Any_Priority := Priority'Last;
package A.C.Bounded_Synchronized_Queues is
  pragma Preelaborate(Bounded_Synchronized_Queues);

  package Implementation is
     ...               -- not specified by the language
  end Implementation;

  protected type Queue(Capacity: Count_Type := Default_Capacity,
                                Ceiling: Any_Priority := Default_Ceiling)
    with Priority => Ceiling
                    is new Queue_Interfaces.Queue with

  ...                          -- etc as for the unbounded one

end A.C.Bounded_Synchronized_Queues;
```

So using the same example, we can use the same interface package Rec_Interface. Now suppose we wish to declare a bounded queue with capacity 1000, we can write

```
package Bounded_Rec_Package is
  new A.C.Bounded_Synchronized_Queues
                    (Queue_Interfaces => Rec_Interface, Default_Capacity => 1000);
```

Finally, we can declare a protected object, My_Rec_BQ which is the actual queue, thus

```
My_Rec_BQ: Bounded_Rec_Package.Queue;
```

And then we can use the queue as before. To place an object on the queue we can write

```
My_Rec_BQ.Enqueue(Some_Rec);
```

And to remove an item from the queue we can write

```
My_Rec_BQ.Dequeue(The_Rec);
```

The major difference is that if the queue becomes full then calling Enqueue will block the calling task until some other task calls Dequeue. Thus, unlike the other containers, Capacity_Error is never raised.

Note that having given a value for Default_Capacity, it can be overridden when the queue is declared, perhaps

```
My_Rec_Giant_BQ: Bounded_Rec_Package.Queue(Capacity => 100000);
```

These packages also provide control over the ceiling priority of the protected type. By default it is Priority'Last. This default can be overridden by our own default when the queue package is instantiated and can be further specified as a discriminant when the actual queue object is declared. So we might write

```
My_Rec_Ceiling_BQ: Bounded_Rec_Package.Queue(Ceiling => 10);
```

In the case of the bounded queue, if we do not give an explicit capacity then the ceiling has to be given using named notation. This does not apply to the unbounded queue which only has one discriminant, so to give that a ceiling priority we can just write

```
My_Rec_Ceiling_UQ: Unbounded_Rec_Package.Queue(10);
```

But clearly the use of the named notation is advisable.

Being able to give default discriminants is very convenient. In Ada 2005, this was not possible if the type was tagged. However, in Ada 2012, it is permitted in the case of limited tagged types and a protected type is considered to be limited. This was explained in detail in the paper on Structure and Visibility.

If we wanted to make a queue of indefinite objects, then as mentioned above, there is no special container for this because Dequeue would be difficult to use since it is a procedure and not a function. So the actual parameter would have to be constrained which means knowing before the call the value of the discriminant, tag, or bound of the object which is unlikely. However, we can use the holder container to wrap the indefinite type so that it looks definite.

So to create a queue for strings, using the example of the previous section, we can write

```
package Strings is
            new Ada.Containers.Indefinite_Holders(String);

package Strings_Interface is
  new A.C.Synchronized_Queue_Interfaces(Element_Type => Strings.Holder);

package Unbounded_Strings_Package is
  new A.C.Unbounded_Synchronized_Queues(Queue_Interfaces => Strings_Interface);
```

and then finally declare the actual queue

```
My_Strings_UQ: Unbounded_Strings_Package.Queue;
```

To put some strings on this queue, we write

```
My_Strings_UQ.Enqueue(To_Holder("rabbit"));

My_Strings_UQ.Enqueue(To_Holder("horse"));
```

or even

```
My_Strings_UQ.Enqueue(Element(Kennel));
```

We now turn to considering the two other forms of queue which are the unbounded and bounded priority queues.

Here is the specification of the unbounded priority queue

```
with System; use System;
with A.C.Synchronized_Queue_Interfaces;
generic
  with package Queue_Interfaces is new
                 A.C.Synchronized_Queue_Interfaces(<>);

  type Queue_Priority is private;
  with function Get_Priority(Element : Queue_Interfaces.Element_Type)
                                                          return Queue_Priority
is <>;
  with function Before(Left, Right : Queue_Priority) return Boolean is <>;

  Default_Ceiling: Any_Priority := Priority'Last;
package A.C.Unbounded_Priority_Queues is
  pragma Preelaborate(Unbounded_Priority_Queues);

  package Implementation is
     ...           -- not specified by the language
  end Implementation;
```

```
   protected type Queue(Ceiling: Any_Priority := Default_Ceiling)
                  with Priority => Ceiling
                    is new Queue_Interfaces.Queue with

      overriding
      entry Enqueue(New_Item: in Queue_Interfaces.Element_Type);
      overriding
      entry Dequeue(Element: out Queue_Interfaces.Element_Type);

      not overriding
      procedure Dequeue_Only_High_Priority(At_Least: in Queue_Priority;
                                           Element: in out
Queue_Interfaces.Element_Type;
                                           Success: out Boolean);

      overriding
      function Current_Use return Count_Type;
      overriding
      function Peak_Use return Count_Type;

   private
      ...            -- not specified by the language
   end Queue;

private
   ...               -- not specified by the language
end A.C.Unbounded_Priority_Queues;
```

The differences from the synchronized bounded queue are that there are several additional generic parameters, namely the private type Queue_Priority and the two functions Get_Priority and Before which operate on objects of the type Queue_Priority, and also that the protected type Queue has an additional operation, the protected procedure Dequeue_Only_High_Priority.

The general idea is that elements have an associated priority which can be ascertained by calling the function Get_Priority. The meaning of this priority is given by the function Before.

When we call Enqueue, the new item is placed in the queue taking due account of its priority with respect to other elements already on the queue. So it will go before all less important elements as defined by Before. If existing elements already have the same priority then it goes after them.

As expected Dequeue just returns the first item on the queue and will block if the queue is empty.

The new procedure Dequeue_Only_High_Priority (note that it is marked as **not overriding** unlike the other operations) is designed to enable us to process items only if they are important enough as defined by the parameter At_Least. The priority of the first element E on the queue is P as given by Get_Priority(E). And so if Before(At_Least, P) is false, then the item on the queue is indeed important enough and so is removed from the queue and the Boolean parameter Success is set to true. On the other hand if Before(At_Least, P) is true then the item is not removed and Success is set to false. Note especially that Dequeue_Only_High_Priority never blocks. If the queue is empty, then Success is just set to false; it never waits for an item to be put on the queue.

As an (unrealistic) example, suppose we decide to make the queue of strings into a priority queue and that the priority is given by their length so that "rabbit" takes precedence over "horse". Remember that the type of the elements is Strings.Holder. We can define the priority as given by the attribute Length so we might as well make the actual type corresponding to Queue_Priority as simply Natural. Then we define

```
function S_Get_Priority(H: Strings.Holder) return Natural is
  (H.Element'Length);

function S_Before(L, R: Natural) return Boolean is
  (L > R);
```

Note the convenient use of expression functions for this sort of thing.

The instantiation now becomes

```
package Unbounded_Priority_Strings_Package is
  new A.C.Unbounded_Priority_Queues(Queue_Interfaces => Strings_Interface,
                                    Queue_Priority => Natural,
                                    Get_Priority => S_Get_Priority,
                                    Before => S_Before);
```

and we then declare a queue thus

```
My_Strings_UPQ: Unbounded_Priority_Strings_Package.Queue;
```

To put some strings on this queue, we write

```
My_Strings_UPQ.Enqueue(To_Holder("rabbit"));

My_Strings_UPQ.Enqueue(To_Holder("horse"));

My_Strings_UPQ.Enqueue(To_Holder("donkey"));

My_Strings_UPQ.Enqueue(To_Holder("gorilla"));
```

The result is that "gorilla" will have jumped to the head of the queue despite having been put on last. It will be followed by "rabbit" and "donkey" and the "horse" is last.

If we do

```
My_Strings_UPQ.Dequeue_Only_High_Priority(7, Kennel, OK);
```

then the "gorilla" will be taken from the queue and placed in the Kennel and OK will be true. But if we then do it again, nothing will happen because the resulting head of the queue (the "rabbit") is not long enough.

Finally, we need to consider bounded priority queues. They are exactly like the unbounded priority queues except that they have the same additional features regarding capacity as found in the synchronized queues. Thus the only differences (apart from the name) are that there is an additional generic parameter Default_Capacity and the protected type Queue has an additional discriminant Capacity.

As a final example we will do a bounded priority queue of records. Suppose the records concern requests for servicing a dishwasher. They might included usual information such as the model number, name and address of owner and so on. They might also have a component indicating degree of urgency, such as

Urgent – machine has vomited dirty water all over floor; housewife/husband having a tantrum,

Major – machine won't do anything; husband refuses to help with washing up,

Minor – machine leaves some dishes unclean, mother-in-law is coming next week,

Routine – machine needs annual service.

So we might have

```
type Degree is (Urgent, Major, Minor, Routine);
type Dish_Job is
  record
    Urgency: Degree;
    Name: ...
    ...
  end record;
```

First we declare the interface for this type

```
package Dish_Interface is
  new A.C.Synchronized_Queue_Interfaces(Element_Type => Dish_Job);
```

and then we declare the two slave functions for the priority mechanism thus

```
function W_Get_Priority(X: Dish_Job) return Degree is
  (X.Urgency);

function W_Before(L, R: Degree) return Boolean is
  (Degree'Pos(L) < Degree'Pos(R));
```

The instantiation is then

```
package Washer_Package is
  new A.C.Bounded_Priority_Queues(Queue_Interfaces => Dish_Interface,
                                  Queue_Priority => Degree,
                                  Get_Priority => W_Get_Priority,
                                  Before => W_Before,
                                  Default_Capacity => 100);
```

and we declare the queue of waiting calls thus

```
Dish_Queue: Washer_Package.Queue;
```

which gives a queue with the default capacity of 100.

The staff taking requests then place the calls on the queue by

```
Dish_Queue.Enqueue(New_Job);
```

To cope with the possibility that the queue is full, they can do a timed entry call; remember that this is possible because the procedure Enqueue in the interface package has Synchronization => By_Entry.

And then general operatives checking in and taking the next job do

```
Dish_Queue.Dequeue(Next_Job);
```

However, at weekends we can suppose that just one operative is on call and deals with only Urgent and Major calls. He might check the queue from time to time by calling

```
Dish_Queue.Dequeue_Only_High_Priority(Major, My_Job, Got_Job);
```

and if Got_Job is false, he can relax and go back to digging the garden or playing golf.

7 Sorting

Ada 2005 provides two containers for sorting arrays; one is for unconstrained array types and one is for constrained array types. The specification of the unconstrained one is

```
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);
```

This does the obvious thing. It sorts the array Container so that the components are in the order defined by the generic parameter "<".

We could for example sort the letters in a string into alphabetical order. We would declare

```
procedure String_Sort is
   new Ada.Containers.Generic_Array_Sort(Positive, Character, String);
```

and then if we had a string such as

```
Bigpet: String := "rabbit";
```

we could apply String_Sort to it thus

```
String_Sort(Bigpet);
```

and the value in Bigpet will now be "abbirt".

That is all in Ada 2005. However, sorting doesn't just apply to arrays and Ada 2012 provides a much more flexible approach. An additional container is provided whose specification is

```
generic
   type Index_Type is (<>);
   with function Before(Left, Right: Index_Type) return Boolean;
   with procedure Swap(Left, Right: in Index_Type);
procedure Ada.Containers.Generic_Sort(First, Last: Index_Type'Base);
pragma Pure(Ada.Containers.Generic_Sort);
```

This can be used to sort any indexable structure and not just arrays. The generic parameters define the required ordering through the parameter Before much as expected. The cunning trick however, is that the means of interchanging two items in the structure is provided by the parameter Swap.

As an illustration we can use this on the array Bigpet. We can use an expression function for BP_Before and so we write

```
function BP_Before(L, R: Positive) return Boolean is
   (Bigpet(L) < Bigpet(R));

procedure BP_Swap(L, R: in Positive) is
   Temp: Character;
begin
   Temp := Bigpet(L);
   Bigpet(L) := Bigpet(R);
   Bigpet(R) := Temp;
end BP_Swap;

procedure BP_Sort is
   new Ada.Containers.Generic_Sort(Positive, BP_Before, BP_Swap);
```

and then we actually do the sort by

```
BP_Sort(Bigpet'First, Bigpet'Last);
```

That may seem to be rather a struggle but the key point is that the technique can be used to sort items in any indexable structure such as a vector container.

Suppose we have a number of records of a type Score which might be

```
type Score is
  record
    N: Natural := 0;
    OS: Other_Stuff;
  end record;
```

and we declare a vector container to hold such objects thus

```
package Scores is
  new Ada.Containers.Vectors(Natural, Score);

My_Vector: Scores.Vector;
```

Now assume that we have added various objects of the type Score to our vector and that we decide that we would like them sorted into order determined by their component N.

We write

```
function MV_Before(L, R: Natural) return Boolean is
  (Scores.Element(My_Vector, L).N < Scores.Element(My_Vector, R).N);

procedure MV_Swap(L, R: in Natural) is
begin
  Scores.Swap(My_Vector, L, R);
end MV_Swap;

procedure MV_Sort is
                    new Ada.Containers.Generic_Sort(Natural, MV_Before, MV_Swap);
```

and then we do the sort by

```
MV_Sort(Scores.First_Index(My_Vector), Scores.Last_Index(My_Vector));
```

Note that the vectors container package conveniently already has a procedure Swap.

This vector example is not very exciting because it might be recalled that the vectors containers already have their own internal generic sort. To use it on this example we would have to write

```
package MV_Sorting is
  new Scores.Generic_Sorting(MV_Before);

MV_Sorting.Sort(My_Vector);
```

which is somewhat simpler. However, note that this sorts the whole vector. If we only wanted to sort part of it, say from elements in index range P to Q then it cannot be used. But that would be easy with the new one since we would simply write

```
MV_Sort(P, Q);
```

Note that curiously this does not need to mention My_Vector.

8 Streaming

Ada 2005 was somewhat unclear regarding streaming values from and to containers. This is clarified in Ada 2012. Thus if V is a vector container then V'Write writes Length(V) elements to the stream concerned.

An important point is that in order to simplify the interchange between containers, we are assured that we can stream between them using 'Write and 'Read. Thus we can stream between a bounded and an unbounded container as well as between two bounded or two unbounded containers provided of course that the elements all have the same subtype.

References

[1] ISO/IEC JTC1/SC22/WG9 N498 (2009) *Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of Amendment 2 to ISO/IEC 8652*.