A brief introduction to Ada 2012

by John Barnes

Chapter 5 - Iterators, Pools, etc.

Courtesy of AdaCore
The GNAT Pro Company

Rationale for Ada 2012: 5 Iterators, Pools, etc.

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 various improvements in a number of general areas in Ada 2012.

There are some minor but perhaps surprising changes concerning matters such as the placement of pragmas and labels.

There are important new features regarding indexing and accessing largely introduced to simplify iterating over containers.

There are also a number of additional Restrictions identifiers many related to the introduction of aspect specifications.

The functionality of access types and storage management is made more flexible by the introduction of subpools.

Finally, a number of minor additions and corrections are made to a range of topics such as generics.

Keywords: rationale, Ada 2012.

1 Overview of changes

The areas mentioned in this paper are not specifically mentioned in the WG9 guidance document [1] other than under the request to remedy shortcomings and improve the functionality of access types and dynamic storage management.

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

6	Nominal subtypes for all names
71	Class-wide operations for formal subprograms
95	Address of intrinsic subprograms
100	Placement of pragmas
111	Subpools, allocators & control of finalization
119	Package Calendar, Daylight Saving Time and UTC_Offset
123	Composability of equality
139	Syntactic sugar for access, containers & iterators
148	Accessibility of anonymous access stand-alone objects
149	Access type conversion and membership
152	Restriction No_Anonymous_Allocators
163	Pragmas in place of null
173	Testing of tags representing abstract types
179	Labels at end of a sequence of statements
189	Restriction No_Standard_Allocators_After_Elaboration
190	Global storage pool control
193	Alignment of allocators
212	Accessors and iterators for Ada.Containers
241	Aspect-related restrictions

242 No_Implementation_Units restriction

246 Restrictions No_Implementation_Identifiers and Profile No_Implementation_Extensions

252 Questions on subpools

253 Accessibility of allocators for anonymous access of an object

255 User-defined iterators and quantified expressions

272 Pragma and attribute restrictions

292 Terminology: indexable type is confusing

These changes can be grouped as follows.

First there are some minor changes to elementary matters such as the placement of pragmas, labels and null statements (100, 163, 179).

An important addition is the introduction of more user-friendly mechanisms for iterating over structures such as arrays and containers (139, 212, 255, 292).

Further flexibility for storage management is provided by the introduction of subpools of storage pools (111, 190, 252). A number of issues concerning anonymous access types and allocators are also resolved (148, 149, 193, 253).

A number of new Restrictions identifiers have been added. They include No_Coextensions, No_Standard_Allocators_After_Elaboration, No_Anonymous_Allocators, No_Implementation_Units, and No_Implementation_Identifiers. A blanket new profile covering a number of restrictions, No_Implementation_Extensions, is also added (152, 189, 241, 242, 246, 272).

Finally, there are a number of minor unrelated improvements. Four are actually classed as binding interpretations and so apply to Ada 2005 as well; they concern nominal subtypes (6), address of intrinsic subprograms (95), time in the package Calendar (119), and class wide operations on formal generic subprograms (71). The other miscellaneous issues concern the composability of equality (123), and tags of abstract types (173).

2 Position of pragmas and labels

It is surprising that basic stuff such as where one can place a pragma should be the subject of discussion thirty years after Ada became an ANSI standard.

However, there is a real problem in this area which one could imagine might have led to headlines in the Wall Street Journal and Financial Times such as

Collapse of NY Stock Market because of Safety Fears in Avionic Applications after Discovery that Ada is Illegal

Indeed, it seems that the package Ada in Ada 2005 might be illegal. This surprising conclusion was triggered by the consideration of

```
task type TT is
  pragma Priority(12);
end TT;
```

The rules in Ada 83, Ada 95 and Ada 2005 concerning the position of pragmas say

Pragmas are only allowed at the following places in a program:

- After a semicolon delimiter, but not within a formal part or discriminant part.
- At any place where the syntax rules allow a construct defined by a syntactic category whose name ends with "declaration", "statement", "clause", or "alternative"; or one of the syntactic

categories variant or exception_handler; but not in place of such a construct. Also at any place where a compilation_unit would be allowed.

Now the syntax for task_definition in Ada 2005 is

```
task_definition ::=
   {task_item}
[private
   {task_item}]
end [task_identifier]
```

There are at least two problems. The key one here is that the list of categories in the rule does not include "item". The other concerns the words "not in place of". It seems that the intent was that if at least one instance of the construct is required (as in a sequence of statements) then the pragma cannot be given in place of a single statement. So it looks as if the task type TT is not legal.

It has probably been permitted because task_item itself splits down into aspect_clause or entry_declaration and they seem to be allowed. But if none is present then we cannot tell which category is permitted!

Note rather scarily that the package Ada is given as

```
package Ada is
  pragma Pure(Ada);
end Ada;
```

and the same problem applies.

The entities in a package specification are of the category basic_declarative_item and again although it splits down into things ending _clause or _declaration we don't know which.

The fear concerning package Ada made one member of the ARG concerned that the sky might be falling in. Of course, we don't ever have to submit a package Ada in our file (on punched cards, paper tape or whatever media we are using). The package Ada is just in the mind of the compiler so that it behaves as if she were declared. The same applies to Standard. They are sort of synthesized and not actually declared.

Anyway, the upshot is that in Ada 2012, the description of the placement of pragmas is corrected by adding "item" to the list and clarifying the meaning of not in place of.

A further discussion considered sequences of statements. In a structure such as an if statement the syntax is

```
if_statement ::=
  if condition then
    sequence_of_statements
  ...
```

where

```
sequence_of_statements ::= statement {statement}
```

The important point is that a sequence_of_statements must have at least one statement. Moreover, the rules for placing pragmas in Ada 2005 do not allow a pragma in place of a construct so we cannot write

```
if B then
  pragma Assert( ... );                  -- illegal in Ada 2005
else ...
```

but have to include at least one statement (such as a null statement) by writing perhaps

```
if B then
  pragma Assert( ... ); null;
else ...
```

or

```
if B then
  null; pragma Assert( ... );
else ...
```

On reflection this seemed irritating so the rules for the placement of pragmas are further amended to include another bullet

- In place of a statement in a sequence_of_statements

A useful note on a language definition principle is added to the AARM which is that if all pragmas are treated as unrecognized then a program should remain legal.

Incidentally, there are other places in the language where at least one item is required such as in a component list. Again if we don't want any components we have to write a null component as in

```
type Nothing is
  record
    null;
  end record;
```

One might have thought that we could similarly now allow one to write

```
type T is
  record
    pragma Page;
  end record;
```

Indeed, it might have been thought that we could simply say that in general a pragma can be given "in place of" an entity. But this doesn't work in some cases. For example, an asynchronous select statement can take the form of a series of statements in its triggering alternative thus

```
select
  S1( ... );
  S2( ... );
  S3( ... );
then abort
  ...
end select;
```

Now the call of S1 is the triggering statement and has a different status to S2 and S3. It would be very confusing to be able to replace the call of S1 by a pragma. So such generalization was dismissed as leading to trouble.

The final topic in this vein concerns the position of labels. This was triggered by the consideration of the problem of quitting one iteration of a loop if it proves unsuccessful and then trying the next iteration. As described in the Introduction this can be done by writing

```
for I in Some_Range loop
  ...
  if not OK then goto End_Of_Loop; end if;
  ...                                  -- lots of other code
```

```
   <<End_Of_Loop> null;              -- try another iteration
   end loop;
```

Of course, maybe we should avoid the goto and write

```
   for I in Some_Range loop
     ...
     if OK then
       ...                           -- lots of other code
     end if;
                                              -- try another iteration
   end loop;
```

At first sight the latter structure looks nicer. However, if the "lots of other code" encounters several situations which mean that the iteration has to be abandoned then we quickly get a deeply nested structure which is not easy to understand and becomes heavily indented.

Much consideration was given to the introduction of a continue statement but it was felt that this would obscure the existence of the transfer of control. Although the goto may be deprecated as obscure, the corresponding obvious label in its aggressive double angle brackets is a strong clue to the existence of the transfer of control.

In the end it was decided that the only sensible improvement was to remove the need for the null statement at the end of the loop.

This is achieved by changing the syntax for a sequence of statements to

```
sequence_of_statements ::= statement {statement} {label}
```

and adding a rule to the effect that if one or more labels end a sequence of statements then an implicit null statement is inserted after the labels. So the loop example can now be written as

```
   for I in Some_Range loop
     ...
     if not OK then goto End_Of_Loop; end if;
     ...                             -- lots of other code
   <<End_Of_Loop>                    -- try another iteration
   end loop;
```

More generally we can write

```
   if B then
     S1; S2; <<My_Label>>
   end if;
```

as well as giving the null explicitly thus

```
   if B then
       S1; S2; <<My_Label>> null;
   end if;
```

but we still cannot write

```
   if B then
     <<My_Label>>                    -- illegal
   end if;
```

since a sequence of statements must still include at least one statement. Of course, we could never jump to such a label anyway since control cannot be transferred into a structure.

3 Iteration

Iteration and subprogram calls are in some sense the twin cornerstones of programming. We are all familiar with the ubiquitous nature of statements such as

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

which in one form or another exist in all (normal) programming languages.

The detail of giving the precise description of the iteration and the indexing is really a violation of abstraction by revealing unnecessary detail. All we want to say is "assign zero to each element of the set A".

However, although it's not too much of a hassle with arrays, the introduction of containers revealed that detailed iteration could be very heavy-handed. Thus, as mentioned in the Introduction, suppose we are dealing with a list, perhaps a list of the type Twin declared as

```
type Twin is
  record
    P, Q: Integer;
  end record;
```

To manipulate every element of the list in Ada 2005, we have to write something like

```
C := The_List.First;                    -- C declared as of type Cursor
loop
  exit when C = No_Element;
  E := Element(C);                      -- E is of type Twin
  if Is_Prime(E.P) then
    Replace_Element(The_List, C, (E.P, E.Q + X));
  end if;
  C := Next(C);
end loop;
```

This reveals the gory details of the iterative process whereas all we want to say is "add X to the component Q for all members of the list whose component P is prime".

There is another way in Ada 2005 and that is to use the procedure Iterate. In that case the details of what we are doing have to be placed in a distinct subprogram called perhaps Do_It. Thus we can write

```
declare
  procedure Do_It(C: in Cursor) is
  begin
    E := Element(C);                    -- E is of type Twin
    if Is_Prime(E.P) then
      Replace_Element(The_List, C, (E.P, E.Q + X));
    end if;
  end Do_It;
begin
  The_List.Iterate(Do_It'Access);
end;
```

This avoids the fine detail of calling First and Next but uses what some consider to be a heavy infrastructure.

However, in Ada 2012 we can simply say

```
for E of The_List loop
  if Is_Prime(E.P) then
    E.Q := E.Q + X;
  end if;
end loop;
```

Not only is this just five lines of text rather than nine or eleven, the key point is that the possibility of making various errors of detail is completely removed.

The mechanisms by which this magic abstraction is achieved are somewhat laborious and it is anticipated that users will take a cookbook approach (show us how to do it, but please don't explain why – after all, this is the approach taken with boiling an egg, we can do it without deep knowledge of the theory of coagulation of protein material).

We will start by looking at the process using arrays. Rather than

```
for I in A'Range loop
  if A(I) /= 0 then
    A(I) := A(I) + 1;
  end if;
end loop;
```

we can write

```
for E of A loop
  if E /= 0 then
    E := E + 1;
  end if;
end loop;
```

In the case of a two-dimensional array, instead of

```
for I in AA'Range(1) loop
  for J in AA'Range(2) loop
    A(I, J) := 0.0;
  end loop;
end loop;
```

we can write

```
for EE of AA loop
  EE := 0.0;
end loop;
```

In Ada 2005 (and indeed in Ada 95 and Ada 83), the syntax for a loop is given by

```
loop_statement ::= [loop_statement_identifier :]
                   [iteration_scheme] loop
                     sequence_of_statements
                   end loop [loop_identifier] ;

iteration_scheme ::= while condition
                      | for loop_parameter_specification

loop_parameter_specification ::= defining_identifier in
                   [reverse] discrete_subtype_definition
```

This is all quite familiar. In Ada 2012, the syntax for loop_statement remains the same but iteration_scheme is extended to give

```
iteration_scheme ::= while condition
                   | for loop_parameter_specification
                   | for iterator_specification
```

Thus the new form iterator_specification is introduced which is

```
iterator_specification ::=
      defining_identifier in [reverse] iterator_name
    | defining_identifier [: subtype_indication] of [reverse] iterable_name
```

The first production defines a *generalized iterator* whereas the second defines an *array component iterator* or a *container element iterator*. For the moment we will just consider the second production which has **of** rather than **in**. The *iterable*_name can refer to an array or a container. Suppose it is an array such as A or AA in the examples above.

We note that we can optionally give the subtype of the loop parameter. Suppose that the type A is given as

```
type A is array (index) of Integer;
```

then the subtype of the loop parameter (E in the example) if not given will just be that of the component which in this case is simply Integer. If we do give the subtype of the loop parameter then it must cover that of the component. This could be useful with tagged types.

Note carefully that the loop parameter does not have the type of the index of the array as in the traditional loop but has the type of the component of the array. So on each iteration it denotes a component of the array. It iterates over all the components of the array as expected. If **reverse** is not specified then the components are traversed in ascending index order whereas if **reverse** is specified then the order is descending. In the case of a multidimensional array then the index of the last dimension varies fastest matching the behaviour of AA in the expanded traditional version as shown (and which incidentally is the order used in streaming). However, if the array has convention Fortran then it is the index of the first dimension that varies fastest both in the case of the loop and in streaming.

There are other obvious rules. If the array A or AA is constant then the loop parameter E or EE is also constant. So it all works much as expected. But do note carefully the use of the reserved word **of** (rather than **is**) which distinguishes this kind of iteration from the traditional form using an index.

As another array example suppose we have the following

```
type Artwin is array (1 .. N) of Twin;

The_Array: Artwin;
```

which is similar to the list example above. In the traditional way we might write

```
for K in Artwin'Range loop
  if Is_Prime(The_Array(K).P) then
    The_Array(K).Q := The_Array(K).Q + X;
  end if;
end loop;
```

Using the new notation this can be simplified to

```
for E: Twin of The_Array loop
  if Is_Prime(E.P) then
    E.Q := E.Q + X;
```

```
   end if;
end loop;
```

where we have added the subtype Twin to clarify the situation. Similarly, in the simple list example we could write

```
for E: Twin of The_List loop
   if Is_Prime(E.P) then
      E.Q := E.Q + X;
   end if;
end loop;
```

Note the beautiful similarity between these two examples. The only lexical difference is that The_Array is replaced by The_List showing that arrays and containers can be treated equivalently.

We now have to consider how the above can be considered as behaving like the original text which involves C of type Cursor, and subprograms First, No_Element, Element, Replace_Element and Next.

This magic is performed by several new features. One is a generic package 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 the various container packages such as Ada.Containers.Doubly_Linked_Lists. Its actual parameters corresponding to the formal parameters Cursor and Has_Element come from the container which includes an instantiation of Ada.Iterator_Interfaces. The instantiation then exports the various required types and functions. Thus in outline the relevant part of the list container now looks like

```
with Ada.Iterator_Interfaces;
generic
   type Element_Type is private;
   with function "=" (Left, Right: Element_Type) return Boolean is <>;
package Ada.Containers.Doubly_Linked_Lists is
   ...
   type List is tagged private ...
   ...
   type Cursor is private;
   ...
   function Has_Element(Position: Cursor) return Boolean;
   package List_Iterator_Interfaces is
      new Ada.Iterator_Interfaces(Cursor, Has_Element);
```

```
  ...
  ...
end Ada.Containers.Doubly_Linked_Lists;
```

The entities exported from the generic package Ada.Iterator_Interfaces are the two interfaces Forward_Iterator and Reversible_Iterator. The interface Forward_Iterator has functions First and Next whereas the Reversible_Iterator (which is itself descended from Forward_Iterator) has functions First and Next inherited from Forward_Iterator plus additional functions Last and Previous.

Note carefully that a Forward_Iterator can only go forward but a Reversible_Iterator can go both forward and backward. Hence it is reversible and not Reverse_Iterator.

The container packages also contain some new functions which return objects of the type Reversible_Iterator'Class or Forward_Iterator'Class. In the case of the list container they are

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

These are new functions and are not to be confused with the existing procedures Iterate and Reverse_Iterate which enable a subprogram to be applied to every element of the list but are somewhat cumbersome to use as shown earlier. The function Iterate with only one parameter is used for iterating over the whole list whereas that with two parameters iterates starting with the cursor value equal to Start.

Now suppose that the list container is instantiated with the type Twin followed by the declaration of a list

```
package Twin_Lists is
  new Ada.Containers.Doubly_Linked_Lists(Element_Type => Twin);

The_List: Twin_Lists.List;
```

So we have now declared The_List which is a list of elements of the type Twin. Suppose we want to do something to every element of the list. As we have seen we might write

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

However, it might be wise at this point to introduce the other from of iterator_specification which is

defining_identifier **in** [**reverse**] *iterator*_name

This defines a generalized iterator and uses the traditional **in** rather than **of** used in the new array component and container element iterators. Using this generalized form we can write

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

In the body of the loop we manipulate the elements using cursors in a familiar way. The reader might wonder why there are these two styles, one using **is** and the other using **of**. The answer is that the generalized iterator is more flexible; for example it does not need to iterate over the whole structure. If we write

```
for C in The_List.Iterate(S) loop
```

then the loop starts with the cursor value equal to S; this is using the version of the function Iterate with two parameters. On the other hand, the array component and container element iterators are more succinct where applicable and are the only from of these new iterators that can be used with arrays.

The generalized iterators for the list container use reversible iterators because the functions Iterate return a value of the type Reversible_Iterator'Class. The equivalent code generated uses the functions First and Next exported from List_Iterator_Interfaces created by the instantiation of Ada.Iterator_Interfaces with the actual parameters The_List.Cursor and The_List.Has_Element. The code then behaves much as if it were (see paragraph 13/3 of subclause 5.5.2 of the RM)

```
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;
```

Of course, the user does not need to know all this in order to use the construction. Note that the functions First and Next used here (which operate on the class Forward_Iterator and are inherited by the class Reversible_Iterator) are not to be confused with the existing functions First and Next which act on the List and Cursor respectively. The existing functions are retained for compatibility and for use in complex situations.

It should also be noted that the initialization of F is legal since the result returned by Iterate is a value of Reversible_Iterator'Class and this is a subclass of Forward_Iterator'Class.

If we had written

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

then the notional code would have been similar but have used the functions Last and Previous rather than First and Next.

Another point is that the function call F.First will deliver the very first cursor value if we had written The_List.Iterate but the value S if we had written The_List.Iterate(S). Remember that we are dealing with interfaces so there is nothing weird here; the two functions Iterate return different types in the class and these have different functions First so the notional generated code calls different functions.

If we use the form

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

then the generated code is essentially the same. However, since we have not explicitly mentioned an iterator, a default one has to be used. This is given by one of several new aspects of the type List which actually now is

```
type List is tagged private
  with Constant_Indexing => Constant_Reference,
            Variable_Indexing => Reference,
            Default_Iterator => Iterate,
            Iterator_Element => Element_Type;
```

The aspect we need at the moment is the one called Default_Iterator which as we see has the value Iterate (this is the one without the extra parameter). So the iterator F is initialized with this default value and once more we get

```
    C: The_List.Cursor;
    E: Twin;
    F: Forward_Iterator'Class := The_List.Iterate;
    ...
```

The use of the other aspects will be explained in a moment.

Lists, vectors and ordered maps and sets can be iterated in both directions. They all have procedures Reverse_Iterate as well as Iterate and the two new functions Iterate return a value of Reversible_Iterator'Class.

However, it might be recalled that the notion of iterating in either direction makes no sense in the case of hashed maps and hashed sets. Consequently, there is no procedure Reverse_Iterate for hashed maps and hashed sets and there is only one new function Iterate which (in the case of hashed maps) is

```
    function Iterate(Container: in Map) return
                              Map_Iterator_Interfaces.Forward_Iterator'Class;
```

and we note that this function returns a value of Forward_Iterator'Class rather than Reversible_Iterator'Class in the case of lists, vectors, ordered maps, and ordered sets.

Naturally, we cannot put **reverse** in an iterator over hashed maps and hashed sets nor can we give a starting value. So the following are both illegal

```
    for C in The_Hash_Map.Iterate(S) loop     -- illegal

    for E of reverse The_Hash_Map loop        -- illegal
```

The above should have given the reader a fair understanding of the mechanisms involved in setting up the loops using the new iterator forms. We now turn to considering the bodies of the loops, that is the code marked "*do something via cursor C* " or "*do something to E* ".

In the Ada 2005 example we wrote

```
      if Is_Prime(E.P) then
        Replace_Element(The_List, C, (E.P, E.Q + X));
      end if;
```

It is somewhat tedious having to write Replace_Element when using a container whereas in the case of an array we might directly write

```
    if Is_Prime(A(I).P) then
      A(I).Q := A(I).Q + X;
    end if;
```

The trouble is that Replace_Element copies the whole new element whereas in the array example we just update the one component. This doesn't matter too much in a case where the components are small such as Twin but if they were giant records it would clearly be a problem. To overcome this Ada 2005 includes a procedure Update_Element thus

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

To use this we have to write a procedure Do_It say thus

```
procedure Do_It(E: in out Twin) is
begin
  E.Q := E.Q + X;
end Do_It;
```

and then

```
  if Is_Prime(E.P) then
    Update_Element(The_List, C, Do_It'Access);
  end if;
```

This works fine because E is passed by reference and no giant copying occurs. However, the downside is that the distinct procedure Do_It has to be written so that the overall text is something like

```
declare
  procedure Do_It(E: in out Twin) is
  begin
    E.Q := E.Q + X;
  end Do_It;
begin
  if Is_Prime(E.P) then
    Update_Element(The_List, C, Do_It'Access);
  end if;
end;
```

which is a bit tedious.

But of course, the text in the body of Do_It is precisely what we want to say. Using the historic concepts of left and right hand values, the problem is that The_List(C).Element cannot be used as a left hand value by writing for example

```
The_List(C).Element.Q := ...
```

The problem is overcome in Ada 2012 using a little more magic by the introduction of generalized reference types and various aspects. In particular we find that the containers now include a type Reference_Type and a function Reference which in the case of the list containers are

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

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

Note the aspect Implicit_Dereference applied to the type Reference_Type with discriminant Element.

There is also a type Constant_Reference_Type and a function Constant_Reference for use when the context demands read-only access.

The alert reader will note the inclusion of **aliased** for the parameter Container of the function Reference. As discussed in the paper on Structure and Visibility, this ensures that the parameter is passed by reference (it always is for tagged types anyway); it also permits us to apply 'Access to the parameter Container within the function and to return that access value.

It might be helpful to say a few words about the possible implementation of Reference and Reference_Type although these need not really concern the user.

The important part of the type Reference_Type is its access discriminant. The private part might contain housekeeping stuff but we can ignore that. So in essence it is simply a record with just one component being the access discriminant

```
type Reference_Type(E: not null access Element_Type) is null record;
```

and the body of the function might be

```
function Reference(Container: aliased in out List;
                    Position: in Cursor) return Reference_Type is
begin
  return (E => Container.Element(Position)'Access);
end Reference;
```

The rules regarding parameters with **aliased** (which we gloss over) ensure that no accessibility problems should arise. Note also that it is important that the discriminant of Reference_Type is an access discriminant since the lifetime of the discriminant is then just that of the return object.

Various aspects are given with the type List which as shown earlier now is

```
type List is tagged private
  with Constant_Indexing => Constant_Reference,
       Variable_Indexing => Reference,
       Default_Iterator => Iterate,
       Iterator_Element => Element_Type;
```

The important aspect here is Variable_Indexing. If this aspect is supplied then in essence the type can be used in a left hand context by invoking the function given as the value of the aspect. In the case of The_List this is the function Reference which returns a value of type Reference_Type. Moreover, this reference type has a discriminant which is of type **access** Element_Type and the aspect Implicit_Dereference with value Element and so gives direct access to the value of type Element.

We can now by stages transform the raw text. So using the cursor form we can start with

```
for C in The_List.Iterator loop
  if Is_Prime(The_List.Reference(C).Element.all.P) then
    The_List.Reference(C).Element.all.Q :=
      The_List.Reference(C).Element.all.Q + X;
  end if;
end loop;
```

This is the full blooded version even down to using **all**.

Omitting the **all** and using the dereferencing with the aspect Implicit_Dereference we can omit the mention of the discriminant Element to give

```
for C in The_List.Iterator loop
  if Is_Prime(The_List.Reference(C).P) then
    The_List.Reference(C).Q := The_List.Reference(C).Q + X;
  end if;
end loop;
```

Remember that Reference is a function with two parameters. It might be clearer to write this without prefix notation which gives

```
for C in Iterator(The_List) loop
  if Is_Prime(Reference(The_List, C).P) then
    Reference(The_List, C).Q := Reference(The_List, C).Q + X;
  end if;
end loop;
```

Now because the aspect Variable_Indexing for the type List has value Reference, the explicit calls of Reference can be omitted to give

```
for C in The_List.Iterator loop
  if Is_Prime(The_List(C).P) then
    The_List(C).Q := The_List(C).Q + X;
  end if;
end loop;
```

It should now be clear that the cursor C is simply acting as an index into The_List. We can compare this text with

```
for C in The_Array'Range loop
  if Is_Prime(The_Array(C).P) then
    The_Array(C).Q := The_Array(C).Q + X;
  end if;
end loop;
```

which shows that 'Range is analogous to .Iterator.

Finally, to convert to the element form using E we just replace The_List(C) by E to give

```
for E of The_List loop
  if Is_Prime(E.P) then
    E.Q := E.Q + X;
  end if;
end loop;
```

The reader might like to consider the transformations in the reverse direction to see how the final succinct form transforms to the expanded form using the various aspects. This is indeed what the compiler has to do.

This underlying technique which transforms the sequence of statements of the container element iterator can be used quite generally. For example, we might not want to iterate over the whole container but just manipulate a particular element given by a cursor C. Rather than calling Update_Element with another subprogram Do_Something, we can write

```
The_List.Reference(C).Q := ...
```

or simply

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

Moreover, although the various aspects were introduced into Ada 2012 primarily to simplify the use of containers they can be used quite generally.

The reader may feel that these new features violate the general ideas of a language with simple building blocks. However, it should be remembered that even the traditional form of loop such as

```
for Index in T range L to U loop
  ...                         -- statements
end loop;
```

is really simply a shorthand for

```
declare
  Index: T;
begin
  if L <= U then
    Index := L;
    loop
      ...                     -- statements
      exit when Index = U;
      Index := T'Succ(Index);
    end loop;
  end if;
end;
```

Without such shorthand, programming would be very tedious and very prone to errors. The features described in this section are simply a further step to make programming safer and simpler.

Further examples of the use of these new features with containers will be given in a later paper dedicated to containers.

The mechanisms discussed above rely on a number of new aspects, a summary of which follows and might be found useful. It is largely based on extracts from the RM.

Dereferencing

The following aspect may be specified for a discriminated type T.

Implicit_Dereference This aspect is specified by a name that denotes an access discriminant of the type T.

A type with a specified Implicit_Dereference aspect is a *reference type*. The Implicit_Dereference aspect is inherited by descendants of type T if not overridden.

A generalized_reference denotes the object or subprogram designated by the discriminant of the reference object.

Indexing

The following aspects may be specified for a tagged type T.

Constant_Indexing This aspect is specified by a name that denotes one or more functions declared immediately within the same declaration list in which T is declared. All such functions shall have at least two parameters, the first of which is of type T or T'Class, or is an access-to-constant parameter with designated type T or T'Class.

Variable_Indexing This aspect is specified by a name that denotes one or more functions declared immediately within the same declaration list in which T is declared. All such functions shall have at least two parameters, the first of which is of type T or T'Class, or is an access parameter with designated type T or T'Class. All such functions shall have a return type that is a reference type, whose reference discriminant is of an access-to-variable type.

These aspects are inherited by descendants of T (including T'Class). The aspects shall not be overridden, but the functions they denote may be.

An *indexable container type* is a tagged type with at least one of the aspects Constant_Indexing or Variable_Indexing specified.

An important difference between Constant_Indexing and Variable_Indexing is that the functions for variable indexing must return a reference type so that it can be used in left hand contexts such as the destination of an assignment. Note that, in both cases, the name can denote several overloaded functions; this is useful, for example, with maps to allow indexing both with cursors and with keys.

Both Constant_Indexing and Variable_Indexing can be provided since the constant one might be more efficient whereas the variable one is necessary in left hand contexts. But we are not obliged to give both, just Variable_Indexing might be enough for some applications.

Iterating

An iterator type is a type descended from the Forward_Iterator interface.

The following aspects may be specified for an indexable container type T.

Default_Iterator This aspect is specified by a name that denotes exactly one function declared immediately within the same declaration list in which T is declared, whose first parameter is of type T or T'Class or an access parameter whose designated type is type T or T'Class, whose other parameters, if any, have default expressions, and whose result type is an iterator type. This function is the *default iterator function* for T.

Iterator_Element This aspect is specified by a name that denotes a subtype. This is the *default element subtype* for T.

These aspects are inherited by descendants of type T (including T'Class).

An *iterable container type* is an indexable container type with specified Default_Iterator and Iterator_Element aspects.

The Constant_Indexing and Variable_Indexing aspects (if any) of an iterable container type T shall denote exactly one function with the following properties:

- the result type of the function is covered by the default element type of T or is a reference type with an access discriminant designating a type covered by the default element type of T;
- the type of the second parameter of the function covers the default cursor type for T;
- if there are more than two parameters, the additional parameters all have default expressions.

These functions (if any) are the *default indexing function*s for T.

The reader might care to check that the aspects used in the examples above match these definitions and are used correctly. Note for example that the Default_Iterator and Iterator_Element aspects are only needed if we use the **of** form of iteration (and both are needed in that case, giving one without the other would be foolish).

This section has largely been about the use of iterators with loop statements. However, there is one other use of them and that is with quantified expressions which are also new to Ada 2012. Quantified expressions were discussed in some detail in the paper on Expressions so all we need here is to consider a few examples which should clarify the use of iterators.

Instead of

```
B := (for all K in A'Range => A(K) = 0);
```

which assigns true to B if every component of the array A has value 0, we can instead write

```
B := (for all E of A  => E = 0);
```

Similarly, instead of

```
B := (for some K in A'Range => A(K) = 0);
```

which assigns true to B if some component of the array A has value 0, we can instead write

```
B := (for some E of A => E = 0);
```

In the case of a multidimensional array, instead of

```
B := (for all I in AA'Range(1) => (for all J in AA'Range(2) => AA(I, J) = 0));
```

we can write

```
B := (for all E of AA => E = 0);
```

which iterates over all elements of the array AA however many dimensions it has.

We can also use these forms with the list example. Suppose we are interested in checking whether some element of the list has a prime component P. We can write

```
B := (for some E of The_List => Is_Prime(E.P));
```

or perhaps

```
B := (for some C in The_List.Iterator => Is_Prime(The_List(C).P));
```

which uses the explicit iterator form.

4 Access types and storage pools

A significant change in Ada 2005 was the introduction of anonymous access types. It is believed that the motivation was to remove the feeling that Ada 95 was unnecessarily pedantic in requiring the introduction of lots of named access types whereas in languages such as C one can just place a star on the identifier of the type being referenced in order to introduce a pointer type.

However, anonymous access types raised more complex accessibility check problems which did not arise with named access types. Most of these problems were resolved in the definition of Ada 2005 but one remained concerning stand-alone objects of anonymous access types. Interestingly, such stand-alone objects were added to Ada 2005 late in the development process; perhaps hastily as it turned out.

In Ada 2005, local stand-alone objects take the accessibility level of the master in which they are declared.

Consider an attempt to use a local stand-alone object in an algorithm to reverse a list. We assume that the list comprises nodes of the following type

```
type Node is
  record
    ...
    Next: access Node;
  end record;
```

and we write

```
function Reverse(List: access Node) return access Node is
  Result: access Node := null;
  This_Node: access Node := List;
  Next_Node: access Node := null;
begin
  while This_Node /= null loop
    Next_Node := This_Node.Next;
    This_Node.Next := Result;            -- access failure in 2005
```

```
      Result := This_Node;
      This_Node := Next_Node;
    end loop;
    return Result;                          --access failure in 2005
  end Reverse;
```

This uses the obvious algorithm of working down the list and rebuilding it. However, in Ada 2005 there are two accessibility failures associated with the variable Result. The assignment to This_Node.Next fails because Result might be referring to something local and we cannot assign that to a node of the list since the list itself lies outside the scope of Reverse_List. Similarly, attempting to return the value in Result fails.

The problem with returning a result can sometimes be solved by using an extended return statement as illustrated in [2]. But this is not a general remedy. The problem is solved in Ada 2012 by treating stand-alone access objects rather like access parameters so that they carry the accessibility of the last value assigned to them as part of their value.

Another reason for introducing anonymous access types in Ada 2005 was to reduce the need for explicit type conversions (note that anonymous access types naturally have no name to use in an explicit conversion). However, it turns out that in practice it is convenient to use anonymous access types in some contexts (such as the component Next of type Node) but in other contexts we might find it logical to use a named access type such as

```
type List is access Node;
```

In Ada 2005, explicit conversions are often required from anonymous access types to named access types and this has been considered to be irritating. Accordingly, the rule has been changed in Ada 2012 to say that an explicit conversion is only required if the conversion could fail.

This relaxation covers both accessibility checks and tag checks. For example we might have

```
type Class_Acc is access T'Class;           -- named type
type Rec is
  record
    Comp: access T'Class;                   -- anon type
  end record;

R: Rec;
```

and then some code somewhere

```
Z: Class_Acc;
...
Z := R.Comp;                                -- OK in Ada 2012
```

The conversion from the anonymous type of Comp to the named type Class_Acc of Z on the assignment to Z cannot fail and so does not require an explicit conversion whereas it did in Ada 2005. Incidentally, the example uses a component Comp rather than a stand-alone object to avoid confusion arising from the special properties of stand-alone objects just discussed.

With regard to tag checks, if it is statically known that the designated type of the anonymous access type is covered by the designated type of the named access type then there is no need for a tag check and so an explicit conversion is not required.

It will be recalled that there is a fictitious type known as *universal_access* (much as *universal_integer*, *root_Integer* and so on). For example, the literal **null** is of this universal type. Moreover, there is a function "=" used to compare *universal_access* values. Permitting implicit conversions requires the introduction of a preference rule for the equality operator of the universal type. Suppose we have

```
type A is access Integer;
R, S: access Integer;
...
if R = S then
```

Now since we can do an implicit conversion from the anonymous access type of R and S to the type A, there is confusion as to whether the comparison uses the equality operator of the type *universal_access* or that of the type A. Accordingly, there is a preference rule that states that in the case of ambiguity there is a preference for equality of the type *universal_access*. Similar preference rules already apply to *root_integer* and *root_real*.

A related topic concerns membership tests which were described in the paper on Expressions.

If we want to ensure that a conversion from perhaps Integer to Index will work and not raise Constraint_Error we can write

```
subtype Index is Integer range 1 .. 20;
I: Index;
K: Integer;
...
if K in Index then
  I := Index(K);                -- bound to work
else
  ...                           -- remedial action
end if;
```

This is much neater than attempting the conversion and then handling Constraint_Error.

However, in Ada 2005, there is no similar facility for testing to see whether an access type conversion would fail. So membership tests in Ada 2012 are extended to permit such a test. So if we have

```
type A is access T1;
X: A;
...
type Rec is
  record
    Comp: access T2;
  end record;

R: Rec;
Y: access T2;
```

we can write

```
if R.Comp in A then
  X := A(R.Comp)                -- conversion bound to work
else ...
```

The membership test will return true if the type T1 covers T2 and the accessibility rules are satisfied so that the conversion is bound to work. Note that the converted expression (R.Comp in this case) can be an access parameter or a stand-alone access object such as Y.

We now turn to consider various features concerning allocation and storage pools.

It will be recalled that if we write our own storage pools then we have to declare a pool type derived from the type Root_Storage_Pool in the package System.Storage_Pools. So we might write

```
package My_Pools is
  type Pond(Size: Storage_Count) is new Root_Storage_Pool with private;
  ...
```

where the discriminant gives the size of the pool. We then have to provide procedures Allocate and Deallocate for our own pool type Pond corresponding to those for Root_Storage_Pool. The procedures Allocate and Deallocate both have four parameters. For example, the procedure Allocate is

```
procedure Allocate(Pool: in out Root_Storage_Pool;
                   Storage_Address: out Address;
                   Size_In_Storage_Elements;
                   Alignment: in Storage_Count) is abstract;
```

When we declare our own Allocate we do not have to use the same names for the formal parameters. So we might more simply write

```
procedure Allocate(Pool: in out Pond;
                   Addr: out Address;
                   SISE: in Storage_Count;
                   Align: in Storage_Count);
```

As well as Allocate and Deallocate we also have to write a function Storage_Size and procedures Initialize and Finalize. However, the key procedures are Allocate and Deallocate which give the algorithms for determining how the storage in the pool is manipulated.

Two parameters of Allocate give the size and alignment of the space to be allocated. However, it is possible that the particular algorithm devised might need to know the worst case values in determining an appropriate strategy. The attribute Max_Size_In_Storage_Elements gives the worst case for the storage size in Ada 2005 but there is no corresponding attribute for the worst case alignment.

This is overcome in Ada 2012 by the provision of the attribute Max_Alignment_For_Allocation. There are various reasons for possibly requiring a different alignment to that expected. For example, the raw objects might simply be byte aligned but the algorithm might decide to append dope or monitoring information which is integer aligned.

The collector of Ada curiosities might remember that Max_Size_In_Storage_Elements is the attribute with most characters in Ada 2005 (28 of which 4 are underlines). Curiously, Max_Alignment_For_Allocation also has 28 characters of which only 3 are underlines.

There are problems with anonymous access types and allocation. Consider

```
package P is
  procedure Proc(X: access Integer);
end P;

with P;
procedure Try_This is
begin
  P.Proc(new Integer'(10));
end Try_This;
```

The procedure Proc has an access parameter X and the call of Proc in Try_This does an allocation with the literal 10. Where does it go? Which pool? Can we do Unchecked_Deallocation? There are special rules for allocators of anonymous access types which aim to answer such questions. The pool is "created at the point of the allocator" and so on.

But various problems arise. An important one is that it is not possible to do unchecked deallocation because the access type has no name; this is particularly serious with library level anonymous access types. An example of such a type might be that of the component Next if the record type Node discussed earlier had been declared at library level.

Consequently, it was concluded that it is best to use named access types if allocation is to be performed. We can always convert to an anonymous type if desired after the allocation has been performed.

In order to avoid encountering such problems a new restriction identifier is introduced. So writing

```
pragma Restrictions(No_Anonymous_Allocators);
```

prevents allocators of anonymous access types and so makes the call of the procedure Proc in the procedure Try_This illegal.

Many long-lived control programs have a start-up phase in which various storage structures are established and which is then followed by the production phase in which various restrictions may be imposed. Ada 2012 has a number of features that enable this to be organized and monitored.

One such feature is the new restriction

```
pragma Restrictions(No_Standard_Allocators_After_Elaboration);
```

This specifies that an allocator using a standard storage pool shall not occur within a parameterless library subprogram or within the statements of a task body. In essence this means that all such allocation must occur during library unit elaboration. Storage_Error is raised if allocation occurs afterwards.

However, it is expected that systems will permit some use of user-defined storage pools. To enable the writers of such pools to monitor their use some additional functions are added to the package Task_Identification so that it now takes the form

```
package Ada.Task_Identification is
   ...
   type Task_Id is private;
   ...
   function Current_Task return Task_Id;
   function Environment_Task return  Task_Id;
   procedure Abort_Task(T: in Task_id);

   function Is_Terminated(T: Task_Id) return Boolean;
   function Is_Callable(T: Task_Id) return Boolean;
   function Activation_Is_Complete(T: Task_Id) return Boolean;
private
   ...
end Ada.Task_Identification;
```

The new function Environment_Task returns the identification of the environment task. The function Activation_Is_Complete returns true if the task concerned has finished activation. Moreover, if Activation_Is_Complete is applied to the environment task then it indicates whether all library items of the partition have been elaborated.

A major new facility is the introduction of subpools. This is an extensive subject so we give only an overview. The general idea is that one wants to manage heaps with different lifetimes. It is often the case that an access type is declared at library level but various groups of objects of the type are declared and so could be reclaimed at a more nested level. This is done by splitting a pool into separately reclaimable subpools. This is far safer and often cheaper than trying to associate lifetimes with individual objects.

A new child package of System.Storage_Pools is declared thus

```
package System.Storage_Pools.Subpools is
  pragma Preelaborate(Subpools);

  type Root_Storage_Pool_With_Subpools is abstract new Root_Storage_Pool with private;

  type Root_Subpool is abstract tagged limited private;

  type Subpool_Handle is access all Root_Subpool'Class;
    for Subpool_Handle'Storage_Size use 0;

  function Create_Subpool(Pool: in out Root_Storage_Pool_With_Subpools)
                                        return not null Subpool_Handle is abstract;

  function Pool_Of_Subpool (Subpool: not null Subpool_Handle) return access
                                              Root_Storage_Pool_With_Subpools'Class;

  procedure Set_Pool_Of_Subpool(Subpool: not null Subpool_Handle;
                                        To: in out
Root_Storage_Pool_With_Subpools'Class);

  procedure Allocate_From_Subpool(Pool: in out Root_Storage_Pool_With_Subpools;
                                  Storage_Address: out Address;
                                  Size_In_Storage_Elements: in Storage_Count;
                                  Alignment: in Storage_Count;
                                  Subpool: in not null Subpool_Handle) is abstract
      with Pre'Class => Pool_Of_Subpool(Subpool) = Pool'Access;

  procedure Deallocate_Subpool(Pool: in out Root_Storage_Pool_With_Subpools;
                               Subpool: in out Subpool_Handle) is abstract
      with Pre'Class => Pool_Of_Subpool(Subpool) = Pool'Access;

  function Default_Subpool_For_Pool(Pool: in out Root_Storage_Pool_With_Subpools)
                                                  return not null Subpool_Handle;

  overriding
  procedure Allocate(Pool: in out Root_Storage_Pool_With_Subpools;
                     Storage_Address: out Address;
                     Size_In_Storage_Elements: in Storage_Count;
                     Alignment: in Storage_Count);

  overriding
  procedure Deallocate( ... ) is null;

  overriding
  function Storage_Size(Pool: Root_Storage_Pool_With_Subpools) return Storage_Count is
          (Storage_Count'Last);

private
  ...        -- not specified by the language
end System.Storage_Pools.Subpools;
```

If we wish to declare a storage pool that can have subpools then rather than declare an object of the type Root_Storage_Pool in the package System.Storage_Pools we have to declare an object of the derived type Root_Storage_Pool_With_Subpools declared in the child package.

The type Root_Storage_Pool_With_Subpools inherits operations Allocate, Deallocate and Storage_Size from the parent type. Remember that Allocate and Deallocate are automatically called by the compiled code when items are allocated and deallocated. In the case of subpools we don't need Deallocate to do anything so it is null. The function Storage_Size determines the value of the attribute Storage_Size and is given by a function expression.

Subpools are separately reclaimable parts of a storage pool and are identified and manipulated by objects of the type Subpool_Handle (these are access values). We can create a subpool by a call of Create_Subpool. So we might have (assuming appropriate with and use clauses)

```
package My_Pools is
  type Pond(Size: Storage_Count) is new Root_Storage_Pool_With_Subpools with private;

  subtype My_Handle is Subpool_Handle;
  ...
```

and then

```
My_Pool: Pond(Size => 1000);

Puddle: My_Handle := Create_Subpool(My_Pool);
```

The implementation of Create_Subpool should call

```
Set_Pool_Of_Subpool(Puddle, My_Pool);
```

before returning the handle. This enables various checks to be made.

In order to allocate an object of type T from a subpool, we have to use a new form of allocator. But first we must ensure that T is associated with the pool itself. So we might write

```
type T_Ptr is access T;
for T_Ptr'Storage_Pool use My_Pool;
```

And then to allocate an object from the subpool identified by the handle Puddle we write

```
X := new (Puddle) T'( ... );
```

where the subpool handle is given in parentheses following **new**.

Of course we don't have to allocate all such objects from a specified subpool since we can still write

```
Y := new T'( ... );
```

and the object will be allocated from the parent pool My_Pool. It is actually allocated from a default subpool in the parent pool and this is determined by writing a suitable body for the function Default_Subpool_For_Pool and this is called automatically by the allocation mechanism. Note that in effect the whole of the pool is divided into subpools one of which may be the default subpool. If we don't provide an overriding body for Default_Subpool_For_Pool then Program_Error is raised. (Note that this function has a parameter of mode **in out** for reasons that need not bother us.)

The implementation carries out various checks. For example, it will check that a handle refers to a subpool of the correct pool by calling the function Pool_Of_Subpool. Both this function and Set_Pool_Of_Subpool are provided by the Ada implementation and typically do not need to be overridden by the implementor of a particular type derived from Root_Storage_Pool_
With_Subpools.

In the case of allocation from a subpool, the procedure Allocate_From_Subpool rather than Allocate is automatically called. Note the precondition to check that all is well.

It will be recalled that for normal storage pools, Deallocate is automatically called from an instance of Unchecked_Deallocation. In the case of subpools the general idea is that we get rid of the whole subpool rather than individual items in it. Accordingly, Deallocate does nothing as mentioned earlier and there is no Deallocate_From_Subpool. Instead we have to write a suitable implementation of Deallocate_Subpool. Note again the precondition to check that the subpool belongs to the pool.

Deallocate_Subpool is called automatically as a consequence of calling the following library procedure

```
with System.Storage_Pools.Subpools;
use System.Storage_Pools.Subpools;
procedure Ada.Unchecked_Deallocate_Subpool(Subpool: in out Subpool_Handle);
```

So when we have finished with the subpool Puddle we can write

```
Unchecked_Dellocate_Subpool(Puddle);
```

and the handle becomes null. Appropriate finalization also takes place.

In summary, the writer of a subpool implementation typically only has to provide Create_Subpool, Allocate_From_Subpool and Deallocate_Subpool since the other subprograms are provided by the Ada implementation of the package System.Storage_Pools.Subpools and can be inherited unchanged.

An example of an implementation will be found in subclause 13.11.6 of the RM. This shows an implementation of a Mark/Release pool in a package MR_Pool. Readers are invited to create variants called perhaps Miss_Pool and Dr_Pool!

Further control over the use of storage pools (nothing to do with subpools) is provided by the ability to define our own default storage pool as mentioned in the Introduction. Thus we can write (and completing our Happy Family of Pools)

```
pragma Default_Storage_Pool(Master_Pool);
```

and then all allocation within the scope of the pragma will be from Master_Pool unless a different specific pool is given for a type. This could be done by using an attribute definition clause thus

```
type Cell_Ptr is access Cell;
  for Cell_Ptr'Storage_Pool use Cell_Ptr_Pool;
```

or by using an aspect specification thus

```
type Cell_Ptr is access Cell
  with Storage_Pool => Cell_Ptr_Pool;
```

A pragma Default_Storage_Pool can be overridden by another one so that for example all allocation in a package (and its children) is from another pool.

The default pool can be specified as **null** thus

```
pragma Default_Storage_Pool(null);
```

and this prevents any allocation from standard pools.

Allocation normally occurs from the default pool unless a specific pool has been given for a type. But there are two exceptions, one concerns access parameter allocation and the other concerns coextensions; in these cases allocation uses a pool that depends upon the context.

Thus in the case of the procedure Proc discussed above, a call such as

```
P.Proc(new Integer'(10));
```

might allocate the space in a secret pool created on the fly and that secret pool might be placed on the stack.

Such allocation can be prevented by two more specific restrictions. They are

```
pragma Restriction(No_Access_Parameter_Allocators);
```

and

```
pragma Restriction(No_Coextensions);
```

These two pragmas plus using the restriction Default_Storage_Pool with **null** ensure that all allocation is from user-defined pools.

5 Restrictions

Restrictions provide a valuable way of increasing security. Ada is a rich language and even richer with Ada 2012 and although individual features are straightforward, certain combinations can cause problems.

The new restrictions introduced into Ada 2012 have already been described in this or earlier papers such as the Introduction. However, for convenience here is a complete list giving the annex where appropriate.

The new Restrictions identifiers are

No_Access_Parameter_Allocators	High-Integrity
No_Anonymous_Allocators	High-Integrity
No_Cooextensions	High-Integrity
No_Implementation_Aspect_Specifications	
No_Implementation_Identifiers	
No_Implementation_Units	
No_Specification_Of_Aspect	
No_Standard_Allocators_After_Elaboration	Real-Time
No_Use_Of_Attribute	
No_Use_Of_Pragma	

Some of the new Restrictions identifiers are in the High-Integrity annex. They are

```
pragma Restrictions(No_Access_Parameter_Allocators);

pragma Restrictions(No_Anonymous_Allocators);

pragma Restrictions(No_Coextensions);
```

and these were discussed in the previous section.

In a similar vein there is one new restriction in the Real-Time annex, namely

```
pragma Restrictions(No_Standard_Allocators_After_Elaboration);
```

and this was also discussed in the previous section.

A number of restrictions prevent the use of implementation-defined features. They are

```
pragma Restrictions(No_Implementation_Aspect_Specifications);

pragma Restrictions(No_Implementation_Identifiers);

pragma Restrictions(No_Implementation_Units);
```

These do not apply to the whole partition but only to the compilation or environment concerned. This helps us to ensure that implementation dependent areas of a program are identified. They were discussed in the Introduction and join similar restrictions No_Implementation_Attributes and No_Implementation_Pragmas introduced in Ada 2005.

The restrictions on implementation-defined aspect specifications, attributes and pragmas are obvious but some clarification of what is meant by the restrictions on units and identifiers might be helpful.

It will be recalled that the predefined packages are Ada, System and Interfaces plus various children. In the so-called standard mode, implementations are not permitted to add their own child packages of Ada but can add grandchildren. Thus an implementation might add an additional container package called perhaps Ada.Containers.Slopbucket. If a program were to use this grandchild then clearly it would be unlikely to be portable to other implementations. Accordingly, giving the restriction No_Implementation_Units prevents such potential difficulties. Similarly, this restriction prevents the use of implementation-defined child units of System and Interfaces.

The restriction No_Implementation_Identifiers is more subtle. It will be recalled that several predefined packages are permitted to add implementation-defined identifiers. They are

Standard, System, Ada.Command_Line, Interfaces.C, Interfaces.C.Strings, Interfaces.C.Pointers, Interfaces.COBOL, and Interfaces.Fortran.

Moreover, the following predefined packages only contain implementation-defined identifiers

Interfaces, System.Machine_Code, Ada.Directories.Information, Ada.Directories.Names, and the packages Implementation nested in the queue containers.

The restriction No_Implementation_Identifiers prevents the use of any of these.

There is a slight subtlety regarding Long_Integer and Long_Float in Standard. The types Integer and Float must be provided. Types such as Short_Integer and Long_Long_Float may be provided but are definitely considered to be implementation-defined and so excluded by the restriction on implementation identifiers. However, Long_Integer and Long_Float should be provided (if the hardware is capable) and so are considered to be predefined and not covered by the restriction. Nevertheless, an implementation on a specialized small machine might not provide them.

Finally, there are restrictions preventing the use of particular facilities

```
pragma Restrictions(No_Specification_Of_Aspect => X);

pragma Restrictions(No_Use_Of_Attribute => X);

pragma Restrictions(No_Use_Of_Pragma => X);
```

where X is the name of a specific aspect, attribute or pragma respectively. They are similar to the restriction No_Dependence introduced in Ada 2005. They apply to a complete partition.

Note that No_Specification_Of_Aspect prevents the specification of an aspect by any means. Remember that some aspects can be specified by an aspect specification or by a pragma or by an attribute definition clause. Thus we mentioned above that a storage pool could be given by an attribute definition clause thus

```
type Cell_Ptr is access Cell;
  for Cell_Ptr'Storage_Pool use Cell_Ptr_Pool;
```

or by using an aspect specification thus

```
type Cell_Ptr is access Cell
  with Storage_Pool => Cell_Ptr_Pool;
```

Writing

```
pragma Restrictions(No_Specification_Of_Aspect => Storage_Pool);
```

prevents both of these whereas

```
pragma Restrictions(No_Use_Of_Attribute => Strorage_Pool);
```

prevents only the first. Naturally, No_Use_Of_Attribute prevents both setting an attribute and using it whereas No_Specification_Of_Aspect prevents just setting it. Thus we might want to use 'Size but prevent setting it.

Similarly

```
pragma Restrictions(No_Specification_Of_Aspect => Pack);
```

prevents both

```
type Flags is array (1 .. 8) of Boolean
  with Pack;
```

and

```
type Flags is array (1 .. 8) of Boolean;
pragma Pack(Flags);
```

whereas

```
pragma Restrictions(No_Use_Of_Pragma => Pack);
```

prevents only the latter.

In summary, No_Specification_Of_Aspect does not mean No_Aspect_Specification (which does not exist).

Remember that several restrictions can be given in one pragma, so we might have

```
pragma Restrictions(No_Use_Of_Pragma => P,
                    No_Use_Of_Attribute  => A);
```

As mentioned in the Introduction there is also a new profile No_Implementation_Extensions. This is specified by

```
pragma Profile(No_Implementation_Extensions);
```

and is equivalent to writing

```
pragma Restrictions(No_Implementation_Aspect_Specifications,
                    No_Implementation_Attributes,
                    No_Implementation_Identifiers,
                    No_Implementation_Pragmas,
                    No_Implementation_Units);
```

thus providing blanket security against writing programs that use language extensions. This profile is defined in the core language. The only other profile defined in Ada 2012 is Ravenscar which was introduced in Ada 2005 and is in the Real-Time systems annex. Remember that the pragma Profile is a configuration pragma.

Finally, those of a recursive nature might note that writing

```
pragma Restrictions(No_Use_Of_Pragma => Restrictions);
```

is illegal (this prevents the risk that the compiler might melt down). More curiously, there is not a restriction No_Implementation_Restrictions. This might be because of similar concern regarding what would happen with its recursive use.

6 Miscellanea

A number of improvements do not neatly fit into any other section of these papers and so are lumped together here.

The first four are in fact binding interpretations and thus apply to Ada 2005 as well.

First, nominal subtypes are defined for enumeration literals and attribute references so that all names now have a nominal subtype.

This is clearly a matter for the language lawyer rather than the happy programmer. Consider the following weird example

```
subtype S is Integer range 1 .. 10;
...
case S'Last is
  when 0 =>   --   ????
```

This is clearly nonsense. However, Ada 2005 does not define a nominal subtype for attributes such as S'Last and so we cannot determine whether 0 is allowed as a discrete choice. The language definition is tidied up to cover such cases.

The second gap in Ada 2005 concerns intrinsic subprograms. Remember that intrinsic subprograms are functions such as "+" on the type Integer that only exist in the mind of the compiler. Clearly they have no address. The following is added to the RM:

> The prefix of X'Address shall not statically denote a subprogram that has convention Intrinsic. X'Address raises Program_Error if X denotes a subprogram that has convention Intrinsic.

The dynamic check is needed because of the possibility of passing an intrinsic operation as a generic parameter.

The third of these binding gems concerns the package Ada.Calendar. The problem is that Calendar.Time is not well-defined when a time zone change occurs as for example when Daylight Saving Time is introduced or removed. Thus operations involving several time values (such as subtraction) might give the "correct" answer or might be an hour adrift. The conclusion reached was simply to admit that it is not defined so the wording is slightly changed.

Another problem with the wording in Ada 2005 is that the sign of the difference between local time and UTC as returned by UTC_Offset is not clearly defined. The sign is clarified so that for example UTC_Offset is negative in the American continent.

There is another problem with the package Calendar which will need to be addressed at some time (probably long after the author is dead). Much effort was exerted in Ada 2005 to cope with leap seconds. These arise because the angular velocity of rotation of the Earth is gradually slowing down. In earlier epochs when measurements of time were not accurate this did not matter. However, we now have atomic clocks and the slowdown is significant so that clocks are adjusted by one second as necessary and these are known as leap seconds.

But leap seconds are under threat. There is a move to suggest that tiny adjustments of one second are not worth the effort and that we should wait until the time is a whole hour wrong. A simple adjustment similar to that with which we are familiar with Daylight Saving changes is all that is needed. In other words we will have a leap hour every now and then. Indeed, if leap seconds occur about once a year as they have done on average since 1972 then a leap hour will be needed sometime in the 37th century. This will probably need to be addressed in Ada 3620 or so.

The final binding interpretation concerns class wide types and generics. An annoyance was recently discovered concerning the use of indefinite container packages such as

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

We can instantiate this with an indefinite type such as String by writing perhaps

```
package String_Vectors is
  new Containers.Indefinite_Vectors(Positive, String);
```

The third actual parameter can be omitted because the predefined operation "=" on the type String exists and does what we want.

Class wide types are another example of indefinite types. Thus we might like to create a vector container whose elements are a mixture of objects of types Circle, Square, Triangle and so on. Assuming these are all descended from the abstract type Object we want to instantiate with the class wide type Object'Class.

However, unlike String, class wide types such as Object'Class do not have a predefined equals. This is annoying since the derived types Circle, Square, and Triangle (being just records) do have a predefined equals.

So we have to write something like

```
function Equal(L, R: Object'Class) is
begin
  return L = R;
end Equal;
```

Note that this will dispatch to the predefined equals of the type of the objects passed as parameters. They both must be of the same type of course; we cannot compare a Circle to a Triangle (anymore than we can compare Thee to a Summer's Day).

So we can now instantiate thus

```
package Object_Vectors is
  new Containers.Indefinite_Vectors(Positive, Object'Class, Equal);
```

Note irritatingly that we cannot write Equal as just "=" because this causes ambiguities.

This is all a bit annoying and so in Ada 2012, the required "=" is automatically created, we do not have to declare Equal, and the instantiation can simply be

```
package Object_Vectors is
  new Containers.Indefinite_Vectors(Positive, Object'Class);
```

This improvement is also a binding interpretation and so applies to Ada 2005 as well.

A more serious matter is the problem of the composability of equality. In Ada 2005, tagged record types compose but untagged record types do not. If we define a new type (a record type, array type or a derived type) then equality is defined in terms of equality for its various components. However, the behaviour of components which are records is different in Ada 2005 according to whether they are tagged or not. If a component is tagged then the primitive operation is used (which might have been redefined), whereas for an untagged type, predefined equality is used even though it might have been overridden.

Consider

```
type Tagrec is tagged
  record
    X1: Integer;
    X2: Integer;
  end record;

type Untagrec is
  record
```

```
    Y1: Integer;
    Y2: Integer;
  end record;

type Index is range 0 .. 64;

...

function "=" (L, R: Tagrec) return Boolean is
begin
  return L.X1 = R.X1;                -- compare only first component
end;

function "=" (L, R: Untagrec) return Boolean is
begin
  return L.Y1 = R.Y1;                -- compare only first component
end;

function "=" (L, R: Index) return Boolean is
begin
  raise Havoc;
  return False;
end;

...

type Mixed is
  record
    T: Tagrec;
    U: Untagrec;
    Z: Index;
  end record;
```

Here we have a type Mixed whose components are of a tagged record type Tagrec, an untagged record type Untagrec, and an elementary type Index. Moreover, we have redefined equality for these types.

In Ada 2005, the equality for the type Mixed uses the redefined equality for the component T but the predefined equality for U and Z. Thus it compares T.X1, U.Y1 and U.Y2 and does not raise Havoc.

In Ada 83, the predefined equality always emerged for the components of arrays and records. One reason was to avoid confusion if an inconsistency arose between "=", "<" and "<=". Remember that many elementary types and certain array types have predefined "<" as well as "=" and to get the relationship messed up would have been confusing.

However, Ada 95 introduced tagged record types and inheritance of operations became an important feature. So it seemed natural that if a structure (array or record) had components of a tagged type and equality for that tagged type had been redefined then it would be natural to expect that equality for the structure should use the redefined equality. But, fearful of introducing an incompatibility, the rule for untagged record types was left unchanged so that predefined equality reemerges.

On reflection, this difference between tagged and untagged records was surprising and so has been changed in Ada 2012 so that all record types behave the same way and use the primitive operation. This is often called composability of equality so we can say that in Ada 2012, record types always compose for equality. Remember that this only applies to records; components which are of array types and elementary types continue to use predefined equality. So in Ada 2012, equality for Mixed only compares T.X1 and U.Y1 but not U.Y2 and still does not raise Havoc.

Concern for incompatibility and inconsistency has been allayed by a deep analysis of a number of programs. No nasties were revealed and in the only cases where it made a difference it was clear that the original behaviour was in fact wrong.

The final miscellaneum (singular of miscellanea?) concerns tags.

The package Ada.Tags defines various functions operating on tags. For example

```
function Parent_Tag(T: Tag) return Tag;
```

returns the tag of the parent unless the type has no parent in which case it returns No_Tag.

However, in Ada 2005 there is no easy way to test whether a tag corresponds to an abstract type. The key property of abstract types is that we cannot have an object of an abstract type. If we wish to create an object using Generic_Dispatching_Constructor and the tag represents an abstract type then Tag_Error is raised. However, it would be far better to check whether a tag represents an abstract type before using Generic_Dispatching_Constructor.

Moreover, if we have a tag and wish to know whether it represents an abstract type, then in Ada 2005 there is no sensible way to find out. We could attempt to create an object and see if it raises Tag_Error. If it doesn't then we know that it was not abstract but we have also created an object we maybe didn't want; if it does raise Tag_Error then it might or might not have been abstract since there are other reasons for the exception being raised. Either way this is madness.

In Ada 2012, we can test the tag using the new function

```
function Is_Abstract(T: Tag) return Boolean;
```

which is added near the end of the package Ada.Tags just before the declaration of the exception Tag_Error.

References

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

[2] John Barnes (2006) *Programming in Ada 2005*, Addison-Wesley.