Rationale for Ada 2005: Introduction

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 is the first of a number of papers describing the rationale for Ada 2005. In due course it is anticipated that the papers will be combined (after appropriate reformatting and editing) into a single volume for formal publication.

This first paper covers the background to the development of Ada 2005 and gives a brief overview of the main changes from Ada 95. Other papers will then look at the changes in more detail.

These papers are being published in the Ada User Journal. An earlier version of this first paper appeared in the Ada User Journal, Vol. 25, Number 4, December 2004. Other papers in this series will be found in later issues of the Journal or elsewhere on this website. The full series is expected to be

0	*Introduction*
1	*Object oriented model*
2	*Access types*
3	*Structure and visibility*
4	*Tasking and Real-Time*
5	*Exceptions, generics etc*
6	*Predefined library*
6a	*Containers*
7	*Epilogue*

Keywords: rationale, Ada 2005.

1 Revision process

Readers will recall that the development of Ada 95 from Ada 83 was an extensive process funded by the USDoD. Formal requirements were established after comprehensive surveys of user needs and competitive proposals were then submitted resulting in the selection of Intermetrics as the developer under the devoted leadership of Tucker Taft. The whole technical development process was then comprehensively monitored by a distinct body of Distinguished Reviewers. Of course, the process was also monitored by the ISO committee concerned and the new language finally became an ISO standard in 1995.

The development of Ada 2005 from Ada 95 has been (and continues to be) on a more modest scale. The work has almost entirely been by voluntary effort with support from within the industry itself through bodies such as the Ada Resource Association and Ada-Europe.

The development is being performed under the guidance of ISO/IEC JTC1/SC22 WG9 (hereinafter just called WG9) chaired adroitly by James Moore whose deep knowledge leads us safely through the minefield of ISO procedures. This committee has included national representatives of many nations including Belgium, Canada, France, Germany, Italy, Japan, Sweden, Switzerland, the UK and the USA. WG9 developed guidelines [1] for a revision to Ada 95 which were then used by the Ada Rapporteur Group (the ARG) in drafting the revised standard.

The ARG is a team of experts nominated by the national bodies represented on WG9 and the two liaison organizations, ACM SIGAda and Ada-Europe. The ARG was originally led with Teutonic precision by Erhard Plödereder and is currently led with Transalpine Gallic flair by Pascal Leroy. The editor, who at the end of the day actually writes the words of the standard, is the indefatigable Randy (fingers) Brukardt.

Suggestions for the revised standard have come from a number of sources such as individuals on the ARG, national bodies on WG9, users via email discussions on Ada-Comment and so on.

At the time of writing (June 2005), the revision process is essentially finished. The details of all individual changes are now clear and they have been integrated to form a new version of the Annotated Ada Reference Manual. This is currently being reviewed and the final approved standard should emerge in the first half of 2006.

There has been much discussion on whether the language should be called Ada 2005 or Ada 2006. For various reasons the WG9 meeting in York in June 2005 decided that the vernacular name should be Ada 2005.

2 Scope of revision

The changes from Ada 83 to Ada 95 were large. They included several major new items such as

_ polymorphism through tagged types, class-wide types and dispatching,

_ the hierarchical library system including public and private child packages,

_ protected objects for better real-time control,

_ more comprehensive predefined library, especially for character and string handling,

_ specialized annexes such as those for system programming, real-time, and numerics.

By contrast the changes from Ada 95 to Ada 2005 are relatively modest. Ada 95 was almost a new language which happened to be compatible with Ada 83. However, a new language always brings surprises and despite very careful design things do not always turn out quite as expected when used in earnest.

Indeed, a number of errors in the Ada 95 standard were corrected in the Corrigendum issued in 2001 [2] and then incorporated into the Consolidated Ada Reference Manual [3]. But it was still essentially the same language and further improvement needed to be done.

Technically, Ada 2005 is defined as an Amendment to rather than a Revision of the Ada 95 standard and this captures the flavour of the changes as not being very extensive.

In a sense we can think of Ada 2005 as rounding out the rough edges in Ada 95 rather than making major leaps forward. This is perhaps not quite true of the Real-Time Systems annex which includes much new material of an optional nature. Nevertheless I am sure that the changes will bring big benefits to users at hopefully not too much cost to implementors.

The scope of the Amendment was guided by a document issued by WG9 to the ARG in September 2002 [1]. The key paragraph is:

"The main purpose of the Amendment is to address identified problems in Ada that are interfering with Ada's usage or adoption, especially in its major application areas (such as high-reliability, longlived real-time and/or embedded applications and very large complex systems). The resulting changes may range from relatively minor, to more substantial."

Note that by saying "identified problems" it implicitly rejects a major redesign such as occurred with Ada 95. The phrase in parentheses draws attention to the areas where Ada has a major market presence. Ada has carved an important niche in the safety-critical areas which almost inevitably are of a real-time and/or embedded nature. But Ada is also in successful use in very large systems where the inherent reliability and composition features are extremely valuable. So changes should aim to help in those areas. And the final sentence is really an exhortation to steer a middle course between too much change and not enough.

The document then identifies two specific worthwhile changes, namely, inclusion of the Ravenscar profile [4] (for predictable real-time) and a solution to the problem of mutually dependent types across two packages (see Section 3.3 below).

The ARG is then requested to pay particular attention to

A Improvements that will maintain or improve Ada's advantages, especially in those user domains where safety and criticality are prime concerns. Within this area it cites as high priority, improvements in the real-time features and improvements in the high integrity features. Of lesser priority are features that increase static error checking. Improvements in interfacing to other languages are also mentioned.

B Improvements that will remedy shortcomings in Ada. It cites in particular improvements in OO features, specifically, adding a Java-like interface feature and improved interfacing to other OO languages.

So the ARG is asked to improve both OO and real-time with a strong emphasis on real-time and high integrity features. It is interesting that WG9 rejected the thought that "design by contract" features should be added to the above general categories on the grounds that they would not be static.

The ARG is also asked to consider the following factors in selecting features for inclusion:

_ Implementability. Can the feature be implemented at reasonable cost?

_ Need. Do users actually need it? [A good one!]

_ Language stability. Would it appear disturbing to current users?

_ Competition and popularity. Does it help to improve the perception of Ada and make it more competitive?

_ Interoperability. Does it ease problems of interfacing with other languages and systems? [That's the third mention of interfacing.]

_ Language consistency. Is it syntactically and semantically consistent with the language's current structure and design philosophy?

An important further statement is that "In order to produce a technically superior result, it is permitted to compromise backwards compatibility when the impact on users is judged to be acceptable." In other words don't be paranoid about compatibility.

Finally, there is a warning about secondary standards. Its essence is don't use secondary standards if you can get the material into the RM itself. And please put the stuff on vectors and matrices from ISO/IEC 13813 [5] into the language itself. The reason for this exhortation is that secondary standards have proved themselves to be almost invisible and hence virtually useless.

The guidelines conclude with the target schedule. This includes WG9 approval of the scope of the amendment in June 2004 which was achieved and submission to ISO/IEC JTC1 in late 2005.

3 Overview of changes

It would be tedious to give a section by section review of the changes as seen by the Reference Manual language lawyer. Instead, the changes will be presented by areas as seen by the user. There can be considered to be six areas:

1 Improvements to the OO model. These include a more traditional notation for invoking an operation of an object without needing to know precisely where the operation is declared (the Obj.Op(...) or prefixed style), Java-like multiple inheritance using the concept of interfaces, the introduction of null procedures as a category of operation rather like an abstract operation, and the ability to do type extension at a more nested level than that of the parent type. There are also explicit features for overcoming nasty bugs that arise from confusion between overloading and overriding.

2 More flexible access types. Ada 95 access types have a hair-shirt flavour compared with other languages because of the general need for explicit conversions with named access types. This is alleviated by permitting anonymous access types in more contexts. It is also possible to indicate whether an access type is an access to a constant and whether a null value is permitted. Anonymous access-to-subprogram types are also introduced thus permitting so-called downward closures.

3 Enhanced structure and visibility control. The most important change here is the introduction of limited with clauses which allow types in two packages to refer to each other (the mutual dependence problem referred to in the WG9 guidelines). This is done by extending the concept of incomplete types (and introducing tagged incomplete types). There are also private with clauses just providing access from a private part. And there are significant changes to limited types to make them more useful; these include initialization using limited aggregates and composition using a new form of return statement.

4 Tasking and real-time improvements. Almost all of the changes are in the Real-Time Systems annex. They include the introduction of the Ravenscar profile (as explicitly mentioned in the WG9 guidelines) and a number of new scheduling and dispatching policies. There are also new predefined packages for controlling execution time clocks and execution time budgets and for the notification of task termination and similar matters. A change related to the OO model is the introduction of protected and task interfaces thereby drawing the OO and tasking aspects of the language closer together.

5 Improvements to exceptions, numerics, generics etc. There are some minor improvements in the exception area, namely, neater ways of testing for null occurrence and raising an exception with a message. Two small but vital numeric changes are a Mod attribute to solve problems of mixing signed and unsigned integers and a fix to the fixed-fixed multiplication problem (which has kept some users locked into Ada 83). There are also a number of new pragmas such as: Unsuppress to complement the Suppress pragma, Assert which was already offered by most vendors, Preelaborable_Initialization which works with the existing pragma Preelaborate, No_Return which indicates that a procedure never returns normally, and Unchecked_Union to ease interfacing to unchecked unions in C. There is also the ability to have more control of partial parameters of generic formal packages to improve package composition.

6 Extensions to the standard library. New packages include a comprehensive Container library, mechanisms for directory operations and access to environment variables, further operations on times and dates, the vectors and matrices material from ISO/IEC 13813 (as directed in the WG9 guidelines) plus commonly required simple linear algebra algorithms. There are also wide-wide character types and operations for 32-bit characters, the ability to use more characters in identifiers, and improvements and extensions to the existing string packages.

Of course, the areas mentioned above interact greatly and much of 2 and 3 could be classified as improvements to the OO model. There are also a number of changes not mentioned which will mostly be of interest to experts in various areas. These cover topics such as streams, object factory functions, subtle aspects of the overload resolution rules, and the categorization of packages with pragmas Pure and Preelaborate.

The reader might feel that the changes are quite extensive but each has an important role to play in making Ada more useful. Indeed many other changes were rejected as really unnecessary. These include old chestnuts such as **in out** and **out** parameters for functions (ugh), extensible enumeration types (a slippery slope), defaults for all generic parameters (would lead one astray), and user-defined operator symbols (a nightmare).

Before looking at the six areas in a little more detail it is perhaps worth saying a few words about compatibility with Ada 95. The guidelines gave the ARG freedom to be sensible in this area. Of course, the worst incompatibilities are those where a valid program in Ada 95 continues to be valid in Ada 2005 but does something different. It is believed that serious incompatibilities of this nature will never arise. There are however, a very few minor and benign such incompatibilities concerning the raising of exceptions such as that with access parameters discussed in Section 3.2.

However, incompatibilities whereby a valid Ada 95 program fails to compile in Ada 2005 are tolerable provided they are infrequent. A few such incompatibilities are possible. The most obvious cause is the introduction of three more reserved words: **interface**, **overriding**, and **synchronized**. Thus if an existing Ada 95 program uses any of these as an identifier then it will need modification. The introduction of a new category of unreserved keywords was considered for these so that incompatibilities would not arise. However, it was felt that this was ugly, confusing, and prone to introducing nasty errors. In any event the identifiers Overriding and Synchronized are likely to be rare and although Interface is clearly a likely identifier nevertheless to have it both as an identifier and as a keyword in the same program would be nasty. Note also that the pragma Interface which many compilers still support from Ada 83 (although not mentioned by Ada 95 at all) is being put into Annex J for obsolescent features.

3.1 The object oriented model

The Ada 95 object oriented model has been criticized as not following the true spirit of the OO paradigm in that the notation for applying subprograms to objects is still dominated by the subprogram and not by the object concerned.

It is claimed that real OO people always give the object first and then the method (subprogram). Thus given

```
package P is
  type T is tagged ... ;

  procedure Op(X: T; ... );
  ...
end P;
```

then assuming that some variable Y is declared of type T, in Ada 95 we have to write

```
P.Op(Y, ... );
```

in order to apply the procedure Op to the object Y whereas a real OO person would expect to write something like

```
Y.Op( ... );
```

where the object Y comes first and only any auxiliary parameters are given in the parentheses.

A real irritation with the Ada 95 style is that the package P containing the declaration of Op has to be mentioned as well. (This assumes that use clauses are not being employed as is often the case.) However, given an object, from its type we can find its primitive operations and it is illogical to require the mention of the package P. Moreover, in some cases involving a complicated type hierarchy, it is not always obvious to the programmer just which package contains the relevant operation.

The prefixed notation giving the object first is now permitted in Ada 2005. The essential rules are that a subprogram call of the form P.Op(Y, ...); can be replaced by Y.Op(...); provided that

- T is a tagged type,
- Op is a primitive (dispatching) or class wide operation of T,
- Y is the first parameter of Op.

The new prefixed notation has other advantages in unifying the notation for calling a function and reading a component of a tagged type. Thus consider the following geometrical example which is based on that in a (hopefully familiar) textbook [6]

```
package Geometry is
  type Object is abstract tagged
    record
      X_Coord: Float;
      Y_Coord: Float;
    end record;

  function Area(O: Object) return Float is abstract;
  function MI(O: Object) return Float is abstract;
end;
```

The type Object has two components and two primitive operations Area and MI (Area is the area of an object and MI is its moment of inertia but the fine details of Newtonian mechanics need not concern us). The key point is that with the new notation we can access the coordinates and the area in a unified way. For example, suppose we derive a concrete type Circle thus

```
package Geometry.Circle is
  type Circle is new Object with
    record
      Radius: Float;
    end record;

  function Area(C: Circle) return Float;
  function MI(C: Circle) return Float;
end;
```

where we have provided concrete operations for Area and MI. Then in Ada 2005 we can access both the coordinates and area in the same way

```
X:= A_Circle.X_Coord;
A:= A_Circle.Area;          -- call of function Area
```

Note that since Area just has one parameter (A_Circle) there are no parentheses required in the call. This uniformity is well illustrated by the body of MI which can be written as

```
  function MI(C: Circle) is
  begin
    return 0.5 * C.Area * C.Radius**2;
  end MI;
```

whereas in Ada 95 we had to write

```
          return 0.5 * Area(C) * C.Radius**2;
```

which is perhaps a bit untidy.

A related advantage concerns dereferencing. If we have an access type such as

```
     type Pointer is access all Object'Class;
     ...
     This_One: Pointer := A_Circle'Access;
```

and suppose we wish to print out the coordinates and area then in Ada 2005 we can uniformly write

```
     Put(This_One.X_Coord); ...
     Put(This_One.Y_Coord); ...
     Put(This_One.Area); ...              -- Ada 2005
```

whereas in Ada 95 we have to write

```
     Put(This_One.X_Coord); ...
     Put(This_One.Y_Coord); ...
     Put(Area(This_One.all)); ...         -- Ada 95
```

In Ada 2005 the dereferencing is all implicit whereas in Ada 95 some dereferencing has to be explicit which is ugly.

The reader might feel that this is all syntactic sugar for the novice and of no help to real macho programmers. So we shall turn to the topic of multiple inheritance. In Ada 95, multiple inheritance is hard. It can sometimes be done using generics and/or access discriminants (not my favourite topic) but it is hard work and often not possible at all. So it is a great pleasure to be able to say that Ada 2005 introduces real multiple inheritance in the style of Java.

The problem with multiple inheritance in the most general case is clashes between the parents. Assuming just two parents, what happens if both parents have the same component (possibly inherited from a common ancestor)? Do we get two copies? And what happens if both parents have the same operation but with different implementations? These and related problems are overcome by placing firm restrictions on the possible properties of parents. This is done by introducing the notion of an interface.

An interface can be thought of as an abstract type with no components – but it can of course have abstract operations. It has also proved useful to introduce the idea of a null procedure as an operation of a tagged type; we don't have to provide an actual body for such a null procedure (and indeed cannot) but it behaves as if it has a body consisting of just a null statement. So we might have

```
     package P1 is
       type Int1 is interface;
       procedure Op1(X: Int1) is abstract;
       procedure N(X: Int1) is null;
     end P1;
```

Note carefully that **interface** is a new reserved word. We could now derive a concrete type from the interface Int1 by

```
       type DT is new Int1 with record ... end record;
       procedure Op1(NX: DT);
```

We can provide some components for DT as shown (although this is optional). We must provide a concrete procedure for Op1 (we wouldn't if we had declared DT itself as abstract). But we do not have to provide an overriding of N since it behaves as if it has a concrete null body anyway (but we could override N if we wanted to).

We can in fact derive a type from several interfaces plus possibly one conventional tagged type. In other words we can derive a tagged type from several other types (the ancestor types) but only one of these can be a normal tagged type (it has to be written first). We refer to the first as the parent (so the parent can be an interface or a normal tagged type) and any others as progenitors (and these have to be interfaces).

So assuming that Int2 is another interface type and that T1 is a normal tagged type then all of the following are permitted

```
type DT1 is new T1 and Int1 with null record;

type DT2 is new Int1 and Int2 with
  record ... end record;

type DT3 is new T1 and Int1 and Int2 with ...
```

It is also possible to compose interfaces to create further interfaces thus

```
type Int3 is interface and Int1;
...
type Int4 is interface and Int1 and Int2 and Int3;
```

Note carefully that **new** is not used in this construction. Such composed interfaces have all the operations of all their ancestors and further operations can be added in the usual way but of course these must be abstract or null.

There are a number of simple rules to resolve what happens if two ancestor interfaces have the same operation. Thus a null procedure overrides an abstract one but otherwise repeated operations have to have the same profile.

Interfaces can also be marked as limited.

```
type LI is limited interface;
```

An important rule is that a descendant of a nonlimited interface must be nonlimited. But the reverse is not true.

Some more extensive examples of the use of interfaces will be given in a later paper.

Incidentally, the newly introduced null procedures are not just for interfaces. We can give a null procedure as a specification whatever its profile and no body is then required or allowed. But they are clearly of most value with tagged types and inheritance. Note in particular that the package Ada.Finalization in Ada 2005 is

```
package Ada.Finalization is
  pragma Preelaborate(Finalization);
  pragma Remote_Types(Finalization);

  type Controlled is abstract tagged private;
  pragma Preeleborable_Initialization(Controlled);
  procedure Initialize(Object: in out Controlled) is null;
  procedure Adjust(Object: in out Controlled) is null;
  procedure Finalize(Object: in out Controlled) is null;

  -- similarly for Limited_Controlled
  ...
end Ada.Finalization;
```

The procedures Initialize, Adjust, and Finalize are now explicitly given as null procedures. This is only a cosmetic change since the Ada 95 RM states that the default implementations have no effect. However, this neatly clarifies the situation and removes ad hoc semantic rules. (The pragma Preelaborable_Initialization will be explained in a later paper.)

Another important change is the ability to do type extension at a level more nested than that of the parent type. This means that controlled types can now be declared at any level whereas in Ada 95, since the package Ada.Finalization is at the library level, controlled types could only be declared at the library level. There are similar advantages in generics since currently many generics can only be instantiated at the library level.

The final change in the OO area to be described here is the ability to (optionally) state explicitly whether a new operation overrides an existing one or not.

At the moment, in Ada 95, small careless errors in subprogram profiles can result in unfortunate consequences whose cause is often difficult to determine. This is very much against the design goal of Ada to encourage the writing of correct programs and to detect errors at compilation time whenever possible. Consider

```
with Ada.Finalization;  use Ada.Finalization;
package Root is
  type T is new Controlled with ... ;
  procedure Op(Obj: in out T; Data: in Integer);
  procedure Finalise(Obj: in out T);
end Root;
```

Here we have a controlled type plus an operation Op of that type. Moreover, we intended to override the automatically inherited null procedure Finalize of Controlled but, being foolish, we have spelt it Finalise. So our new procedure does not override Finalize at all but merely provides another operation. Assuming that we wrote Finalise to do something useful then we will find that nothing happens when an object of the type T is automatically finalized at the end of a block because the inherited null procedure is called rather than our own code. This sort of error can be very difficult to track down.

In Ada 2005 we can protect against such errors since it is possible to mark overridding operations as such thus

```
  overriding
  procedure Finalize(Obj: in out T);
```

And now if we spell Finalize incorrectly then the compiler will detect the error. Note that **overriding** is another new reserved word. However, partly for reasons of compatibility, the use of overriding indicators is optional; there are also deeper reasons concerning private types and generics which will be discussed in a later paper.

Similar problems can arise if we get the profile wrong. Suppose we derive a new type from T and attempt to override Op thus

```
package Root.Leaf is
  type NT is new T with null record;
  procedure Op(Obj: in out NT; Data: in String);
end Root.Leaf;
```

In this case we have given the identifier Op correctly but the profile is different because the parameter Data has inadvertently been declared as of type String rather than Integer. So this new version of Op will simply be an overloading rather than an overriding. Again we can guard against this sort of error by writing

```
overriding
procedure Op(Obj: in out NT; Data: in Integer);
```

On the other hand maybe we truly did want to provide a new operation. In this case we can write **not overriding** and the compiler will then ensure that the new operation is indeed not an overriding of an existing one thus

```
not overriding
procedure Op(Obj: in out NT; Data: in String);
```

The use of these overriding indicators prevents errors during maintenance. Thus if later we add a further parameter to Op for the root type T then the use of the indicators will ensure that we modify all the derived types appropriately.

3.2 Access types

It has been said that playing with pointers is like playing with fire – properly used all is well but carelessness can lead to disaster. In order to avoid disasters, Ada 95 takes a stern view regarding the naming of access types and their conversion. However, experience has shown that the Ada 95 view is perhaps unnecessarily stern and leads to tedious programming.

We will first consider the question of giving names to access types. In Ada 95 all access types are named except for access parameters and access discriminants. Thus we might have

```
type Animal is tagged
  record Legs: Integer; ... end record;

type Acc_Animal is access Animal;                -- named

procedure P(Beast: access Animal; ... );         -- anonymous
```

Moreover, there is a complete lack of symmetry between named access types and access parameters. In the case of named access types, they all have a null value (and this is the default on declaration if no initial value be given). But in the case of access parameters, a null value is not permitted as an actual parameter. Furthermore, named access types can be restricted to be access to constant types such as

```
type Rigid_Animal is access constant Animal;
```

which means that we cannot change the value of the Animal referred to. But in the case of access parameters, we cannot say

```
procedure P(Beast: access constant Animal);            -- not 95
```

In Ada 2005 almost all these various restrictions are swept away in the interests of flexibility and uniformity.

First of all we can explicitly specify whether an access type (strictly subtype) has a null value. We can write

```
type Acc_Animal is not null access all Animal'Class;
```

This means that we are guaranteed that an object of type Acc_Animal cannot refer to a null animal. Therefore, on declaration such an object should be initialized as in the following sequence

```
type Pig is new Animal with ... ;
Empress_Of_Blandings: aliased Pig := ... ;

My_Animal: Acc_Animal := Empress_Of_Blandings'Access;            -- must initialize
```

(The Empress of Blandings is a famous pig in the novels concerning Lord Emsworth by the late P G Wodehouse.) If we forget to initialize My_Animal then Constraint_Error is raised; technically the underlying type still has a null value but Acc_Animal does not. We can also write **not null access constant** of course.

The advantage of using a null exclusion is that when we come to do a dereference

```
Number_of_Legs: Integer := My_Animal.Legs;
```

then no check is required to ensure that we do not dereference a null pointer. This makes the code faster.

The same freedom to add **constant** and **not null** also applies to access parameters. Thus we can write all of the following in Ada 2005

```
procedure P(Beast: access Animal);
procedure P(Beast: access constant Animal);

procedure P(Beast: not null access Animal);
procedure P(Beast: not null access constant Animal);
```

Note that **all** is not permitted in this context since access parameters always are general (that is, they can refer to declared objects as well as to allocated ones).

Note what is in practice a minor incompatibility, the first of the above now permits a null value as actual parameter in Ada 2005 whereas it was forbidden in Ada 95. This is actually a variation at runtime which is normally considered abhorrent. But in this case it just means that any check that will still raise Constraint_Error will be in a different place – and in any event the program was presumably incorrect.

Another change in Ada 2005 is that we can use anonymous access types other than just as parameters (and discriminants). We can in fact also use anonymous access types in

_ the declaration of stand-alone objects and components of arrays and records,

_ a renaming declaration,

_ a function return type.

Thus we can extend our farmyard example

```
type Horse is new Animal with ... ;

type Acc_Horse is access all Horse;
type Acc_Pig is access all Pig;

Napoleon, Snowball: Acc_Pig := ... ;

Boxer, Clover: Acc_Horse := ... ;
```

and now we can declare an array of animals

```
Animal_Farm: constant array (Positive range <>) of access Animal'Class :=
                                               (Napoleon, Snowball, Boxer, Clover);
```

(With acknowledgments to George Orwell.) Note that the components of the array are of an anonymous access type. We can also have record components of an anonymous type

```
type Ark is
  record
    Stallion, Mare: access Horse;
    Boar, Sow: access Pig;
    Cockerel, Hen: access Chicken;
    Ram, Ewe: access Sheep;
    ...
  end record;

Noahs_Ark: Ark := (Boxer, Clover, ... );
```

This is not a very good example since I am sure that Noah took care to take actual animals into the Ark and not merely their addresses.

A more useful example is given by the classic linked list. In Ada 95 (and Ada 83) we have

```
type Cell;
type Cell_Ptr is access Cell;

type Cell is
  record
    Next: Cell_Ptr;
    Value: Integer;
  end record;
```

In Ada 2005, we do not have to declare the type Cell_Ptr in order to declare the type Cell and so we do not need to use the incomplete declaration to break the circularity. We can simply write

```
type Cell is
  record
    Next: access Cell;
    Value: Integer;
  end record;
```

Here we have an example of the use of the type name Cell within its own declaration. In some cases this is interpreted as referring to the current instance of the type (for example, in a task body) but the rule has been changed to permit its usage as here.

We can also use an anonymous access type for a single variable such as

```
List: access Cell := ... ;
```

An example of the use of an anonymous access type for a function result might be in another animal function such as

```
function Mate_Of(A: access Animal'Class) return access Animal'Class;
```

We could then perhaps write

```
if Mate_Of(Noahs_Ark.Ram) /= Noahs_Ark.Ewe then
  ... -- better get Noah to sort things out
end if;
```

Anonymous access types can also be used in a renaming declaration. This and other detailed points on matters such as accessibility will be discussed in a later paper.

The final important change in access types concerns access to subprogram types. Access to subprogram types were introduced into Ada 95 largely for the implementation of callback. But important applications of such types in other languages (going back to Pascal and even Algol 60) are for mathematical applications such as integration where a function to be manipulated is passed as a parameter. The Ada 83 and Ada 95 approach has always been to say "use generics". But this can be clumsy and so a direct alternative is now provided.

Recall that in Ada 95 we can write

```
type Integrand is access function(X: Float) return Float;
```

```
function Integrate(Fn: Integrand; Lo, Hi: Float) return Float;
```

The idea is that the function Integrate finds the value of the integral of the function passed as parameter Fn between the limits Lo and Hi. This works fine in Ada 95 for simple cases such as where the function is declared at library level. Thus to evaluate

$$\int_0^1 \sqrt{x}\, dx$$

we can write

```
Result := Integrate(Sqrt'Access, 0.0, 1.0);
```

where the function Sqrt is from the library package Ada.Numerics.Elementary_Functions.

However, if the function to be integrated is more elaborate then we run into difficulties in Ada 95 if we attempt to use access to subprogram types. Consider the following example which aims to compute the integral of the expression xy over the square region $0 \le x, y \le 1$.

```
with Integrate;
procedure Main is
  function G(X: Float) return Float is
    function F(Y: Float) return Float is
    begin
      return X*Y;
    end F;
  begin
    return Integrate(F'Access, 0.0, 1.0);      -- illegal in 95
  end G;

  Result: Float;
begin
  Result:= Integrate(G'Access, 0.0, 1.0);      -- illegal in 95
  ...
end Main;
```

But this is illegal in Ada 95 because of the accessibility rules necessary with named access types in order to prevent dangling references. Thus we need to prevent the possibility of storing a pointer to a local subprogram in a global structure. This means that both F'Access and G'Access are illegal in the above.

Note that although we could make the outer function G global so that G'Access would be allowed nevertheless the function F has to be nested inside G in order to gain access to the parameter X of G. It is typical of functions being integrated that they have to have information passed globally – the number of parameters of course is fixed by the profile used by the function Integrate.

The solution in Ada 2005 is to introduce anonymous access to subprogram types by analogy with anonymous access to object types. Thus the function Integrate becomes

```
function Integrate(Fn: access function(X: Float) return Float;
                   Lo, Hi: Float) return Float;
```

Note that the parameter Fn has an anonymous type defined by the profile so that we get a nesting of profiles. This may seem a bit convoluted but is much the same as in Pascal.

The nested example above is now valid and no accessibility problems arise. (The reader will recall that accessibility problems with anonymous access to object types are prevented by a runtime check; in the case of anonymous access to subprogram types the corresponding problems are prevented by decreeing that the accessibility level is infinite – actually the RM says larger than that of any master which comes to the same thing.)

Anonymous access to subprogram types are also useful in many other applications such as iterators as will be illustrated later.

Note that we can also prefix all access to subprogram types, both named and anonymous, by **constant** and **not null** in the same way as for access to object types.

3.3 Structure, visibility, and limited types

Structure is vital for controlling visibility and thus abstraction. There were huge changes in Ada 95. The whole of the hierarchical child unit mechanism was introduced with both public and private children. It was hoped that this would provide sufficient flexibility for the future.

But one problem has remained. Suppose we have two types where each wishes to refer to the other. Both need to come first! Basically we solve the difficulty by using incomplete types. We might have a drawing package concerning points and lines in a symmetric way. Each line contains a list or array of the points on it and similarly each point contains a list or array of the lines through it. We can imagine that they are both derived from some root type containing printing information such as color. In Ada 95 we might write

```
type Object is abstract tagged
  record
    Its_Color: Color;
    ...
  end record;

type Point;
type Line;
type Acc_Point is access all Point;
type Acc_Line is access all Line;

subtype Index is Integer range 0 .. Max;
type Acc_Line_Array is array (1 .. Max) of Acc_Line;
type Acc_Point_Array is array (1 .. Max) of Acc_Point;

type Point is new Object with
  record
    No_Of_Lines: Index;
    LL: Acc_Line_Array;
    ...
  end record;

type Line is new Object with
  record
    No_Of_Points: Index;
    PP: Acc_Point_Array;
    ...
  end record;
```

This is very crude since it assumes a maximum number Max of points on a line and vice versa and declares the arrays accordingly. The reader can flesh it out more flexibly. Well this is all very well but if the individual types get elaborate and each has a series of operations, we might want to declare them in distinct packages (perhaps child packages of that containing the root type). In Ada 95 we cannot do this because both the incomplete declaration and its completion have to be in the same package.

The net outcome is that we end up with giant cumbersome packages.

What we need therefore is some way of logically enabling the incomplete view and the completion to be in different packages. The elderly might remember that in the 1980 version of Ada the situation was even worse – the completion had to be in the same list of declarations as the incomplete declaration. Ada 83 relaxed this (the so-called Taft Amendment) and permits the private part and body to be treated as one list – the same rule applies in Ada 95. We now go one step further.

Ada 2005 solves the problem by introducing a variation on the with clause – the limited with clause. The idea is that a library package (and subprogram) can refer to another library package that has not yet been declared and can refer to the types in that package but only as if they were incomplete types. Thus we might have a root package Geometry containing the declarations of Object, Max, Index, and so on and then

```
limited with Geometry.Lines;
package Geometry.Points is

  type Acc_Line_Array is array (1 .. Max) of access Lines.Line;

  type Point is new Object with
    record
      No_Of_Lines: Index;
      LL: Acc_Line_Array;
      ...
    end record;

  ...
end Geometry.Points;
```

The package Geometry.Lines is declared in a similar way. Note especially that we are using the anonymous access type facility discussed in Section 3.2 and so we do not even have to declare named access types such as Acc_Line in order to declare Acc_Line_Array.

By writing **limited with** Geometry.Lines; we get access to all the types visible in the specification of Geometry.Lines but as if they were declared as incomplete. In other words we get an incomplete view of the types. We can then do all the things we can normally do with incomplete types such as use them to declare access types. (Of course the implementation checks later that Geometry.Lines does actually have a type Line.)

Not only is the absence of the need for a named type Acc_Line a handy shorthand, it also prevents the proliferation of named access types. If we did want to use a named type Acc_Line in both packages then we would have to declare a distinct type in each package. This is because from the point of view of the package Points, the Acc_Line in Lines would only be an incomplete type (remember each package only has a limited view of the other) and thus would be essentially unusable. The net result would be many named access types and wretched type conversions all over the place.

There are also some related changes to the notation for incomplete types. We can now write

```
type T is tagged;
```

and we are then guaranteed that the full declaration will reveal T to be a tagged type. The advantage is that we also know that, being tagged, objects of the type T will be passed by reference. Consequently we can use the type T for parameters before seeing its full declaration. In the example of points and lines above, since Line is visibly tagged in the package Geometry.Lines we will thus get an incomplete tagged view of Lines.

The introduction of tagged incomplete types clarifies the ability to write

```
type T_Ptr is access all T'Class;
```

This was allowed in Ada 95 even though we had not declared T as tagged at this point. Of course it implied that T would be tagged. In Ada 2005 this is frowned upon since we should now declare that T is tagged incomplete if we wish to declare a class wide access type. For compatibility the old feature has been retained but banished to Annex J for obsolescent features.

Further examples of the use of limited with clauses will be given in a later paper.

Another enhancement in this area is the introduction of private with clauses which overcome a problem with private child packages.

Private child packages were introduced to enable the details of the implementation of part of a system to be decomposed and yet not be visible to the external world. However, it is often convenient to have public packages that use these details but do not make them visible to the user. In Ada 95 a parent or sibling body can have a with clause for a private child. But the specifications cannot. These rules are designed to ensure that information does not leak out via the visible part of a specification. But there is no logical reason why the private part of a package should not have access to a private child. Ada 2005 overcomes this by introducing private with clauses. We can write

```
private package App.Secret_Details is
  type Inner is ...
  ...        -- various operations on Inner etc
end App.Secret_Details;

private with App.Secret_Details;
package App.User_View is

  type Outer is private;
  ...        -- various operations on Outer visible to the user

   -- type Inner is not visible here
private
   -- type Inner is visible here

   type Outer is
    record
      X: Secret_Details.Inner;
      ...
    end record;
  ...
end App.User_View;
```

thus the private part of the public child has access to the type Inner but it is still hidden from the external user.

Note that the public child and private child might have mutually declared types as well in which case they might also wish to use the limited with facility. In this case the public child would have a limited private with clause for the private child written thus

```
limited private with App.Secret_Details;
package App.User_View is ...
```

In the case of a parent package, its specification cannot have a with clause for a child – logically the specification cannot know about the child because the parent must be declared (that is put into the program library) first. Similarly a parent cannot have a private with clause for a private child. But it can have a limited with clause for any child (thereby breaking the circularity) and in particular it can have a limited private with clause for a private child. So we might also have

```
limited private with App.Secret_Details;
package App is ...
```

The final topic in this section is limited types. The reader will recall that the general idea of a limited type is to restrict the operations that the user can perform on a type to just those provided by the developer of the type and in particular to prevent the user from doing assignment and thus making copies of an object of the type.

However, limited types have never quite come up to expectation both in Ada 83 and Ada 95. Ada 95 brought significant improvements by disentangling the concept of a limited type from a private type but problems have remained.

The key problem is that Ada 95 does not allow the initialization of limited types because of the view that initialization requires assignment and thus copying. A consequence is that we cannot declare constants of a limited type either. Ada 2005 overcomes this problem by allowing initialization by aggregates.

As a simple example, consider

```
type T is limited
  record
    A: Integer;
    B: Boolean;
    C: Float;
  end record;
```

in which the type as a whole is limited but the components are not. If we declare an object of type T in Ada 95 then we have to initialize the components (by assigning to them) individually thus

```
  X: T;
begin
  X.A := 10;  X.B := True;  X.C := 45.7;
```

Not only is this annoying but it is prone to errors as well. If we add a further component D to the record type T then we might forget to initialize it. One of the advantages of aggregates is that we have to supply all the components (allowing automatic so-called full coverage analysis, a key benefit of Ada).

Ada 2005 allows the initialization with aggregates thus

```
  X: T := (A => 10,  B => True,  C => 45.7);
```

Technically, Ada 2005 just recognizes properly that initialization is not assignment. Thus we should think of the individual components as being initialized individually *in situ* – an actual aggregated value is not created and then assigned. (Much the same happens when initializing controlled types with an aggregate.)

Sometimes a limited type has components where an initial value cannot be given. This happens with task and protected types. For example

```
protected type Semaphore is ... ;

type PT is
  record
    Guard: Semaphore;
    Count: Integer;
```

```
   Finished: Boolean := False;
 end record;
```

Remember that a protected type is inherently limited. This means that the type PT is limited because a type with a limited component is itself limited. It is good practice to explicitly put **limited** on the type PT in such cases but it has been omitted here for illustration. Now we cannot give an explicit initial value for a Semaphore but we would still like to use an aggregate to get the coverage check. In such cases we can use the box symbol <> to mean use the default value for the type (if any). So we can write

```
X: PT := (Guard => <>, Count => 0, Finished => <>);
```

Note that the ability to use <> in an aggregate for a default value is not restricted to the initialization of limited types. It is a new feature applicable to aggregates in general. But, in order to avoid confusion, it is only permitted with named notation.

Limited aggregates are also allowed in other similar contexts where copying is not involved including as actual parameters of mode **in**.

There are also problems with returning results of a limited type from a function. This is overcome in Ada 2005 by the introduction of an extended form of return statement. This will be described in detail in a later paper.

3.4 Tasking and real-time facilities

Unless mentioned otherwise all the changes in this section concern the Real-Time Systems annex.

First, the well-established Ravenscar profile is included in Ada 2005 as directed by WG9. A profile is a mode of operation and is specified by the pragma Profile which defines the particular profile to be used. Thus to ensure that a program conforms to the Ravenscar profile we write

```
pragma Profile(Ravenscar);
```

The purpose of Ravenscar is to restrict the use of many of the tasking facilities so that the effect of the program is predictable. This is very important for real-time safety-critical systems. In the case of Ravenscar the pragma is equivalent to the joint effect of the following pragmas

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
pragma Locking_Policy(Ceiling_Locking);
pragma Detect_Blocking;
```

plus a **pragma** Restrictions with a host of arguments such as No_Abort_Statements and No_Dynamic_Priorities.

The pragma Detect_Blocking plus many of the Restrictions identifiers are new to Ada 2005. Further details will be given in a later paper.

Ada 95 allows the priority of a task to be changed but does not permit the ceiling priority of a protected object to be changed. This is rectified in Ada 2005 by the introduction of an attribute Priority for protected objects and the ability to change it by a simple assignment such as

```
My_PO'Priority := P;
```

inside a protected operation of the object My_PO. The change takes effect at the end of the protected operation.

The monitoring and control of execution time naturally are important for real-time programs. Ada 2005 includes packages for three different aspects of this

Ada.Execution_Time – this is the root package and enables the monitoring of execution time of individual tasks.

Ada.Execution_Time.Timers – this provides facilities for defining and enabling timers and for establishing a handler which is called by the run time system when the execution time of the task reaches a given value.

Ada.Execution_Time.Group_Budgets – this allows several tasks to share a budget and provides means whereby action can be taken when the budget expires.

The execution time of a task or CPU time, as it is commonly called, is the time spent by the system executing the task and services on its behalf. CPU times are represented by the private type CPU_Time. The CPU time of a particular task is obtained by calling the following function Clock in the package Ada.Execution_Time

```
function Clock(T: Task_Id := Current_Task) return CPU_Time;
```

A value of type CPU_Time can be converted to a Seconds_Count plus residual Time_Span by a procedure Split similar to that in the package Ada.Real_Time. Incidentally we are guaranteed that the granularity of CPU times is no greater than one millisecond and that the range is at least 50 years.

In order to find out when a task reaches a particular CPU time we use the facilities of the child package Ada.Execution_Time.Timers. This includes a discriminated type Timer and a type Handler thus

```
type Timer(T: not null access constant Task_Id) is tagged limited private;
type Handler is access protected procedure (TM: in out Timer);
```

Note how the access discriminant illustrates the use of both **not null** and **constant**.

We can then set the timer to expire at some absolute time by

```
Set_Handler(My_Timer, Time_Limit, My_Handler'Access);
```

and then when the CPU time of the task reaches Time_Limit (of type CPU_Time), the protected procedure My_Handler is executed. Note how the timer object incorporates the information regarding the task concerned using an access discriminant and that this is passed to the handler via its parameter. Another version of Set_Handler enables the timer to be triggered after a given interval (of type Time_Span).

In order to program various aperiodic servers it is necessary for tasks to share a CPU budget. This can be done using the child package Ada.Execution_Time.Group_Budgets. In this case we have

```
type Group Budget is tagged limited private;
type Handler is access protected procedure (GB: in out Group_Budget);
```

The type Group_Budget both identifies the group of tasks it belongs to and the size of the budget. Various subprograms enable tasks to be added to and removed from a group budget. Other procedures enable the budget to be set and replenished.

A procedure Set_Handler associates a particular handler with a budget.

```
Set_Handler(GB => My_Group_Budget, Handler => My_Handler'Access);
```

When the group budget expires the associated protected procedure is executed.

A somewhat related topic is that of low level timing events. The facilities are provided by the package Ada.Real_Time.Timing_Events. In this case we have

```
type Timing_Event is tagged limited private;
type Timing_Event_Handler is access protected procedure (Event: in out Timing_Event);
```

The idea here is that a protected procedure can be nominated to be executed at some time in the future. Thus to ring a pinger when our egg is boiled after four minutes we might have a protected procedure

```
protected body Egg is
  procedure Is_Done(Event: in out Timing_Event) is
  begin
    Ring_The_Pinger;
  end Is_Done;
end Egg;
```

and then

```
Egg_Done: Timing_Event;
Four_Min: Time_Span := Minutes(4);
...
Put_Egg_In_Water;
Set_Handler(Event => Egg_Done, In_Time => Four_Min, Handler => Egg.Is_Done'Access);
-- now read newspaper whilst waiting for egg
```

This facility is of course very low level and does not involve Ada tasks at all. Note that we can set the event to occur at some absolute time as well as at a relative time as above. Incidentally, the function Minutes is a new function added to the parent package Ada.Real_Time. Otherwise we would have had to write something revolting such as 4*60*Milliseconds(1000). A similar function Seconds has also been added.

There is a minor flaw in the above example. If we are interrupted by the telephone between putting the egg in the water and setting the handler then our egg will be overdone. We will see how to cure this in a later paper.

Readers will recall the old problem of how tasks can have a silent death. If something in a task goes wrong in Ada 95 and an exception is raised which is not handled by the task, then it is propagated into thin air and just vanishes. It was always deemed impossible for the exception to be handled by the enclosing unit because of the inherent asynchronous nature of the event.

This is overcome in Ada 2005 by the package Ada.Task_Termination which provides facilities for associating a protected procedure with a task. The protected procedure is invoked when the task terminates with an indication of the reason. Thus we might declare a protected object Grim_Reaper

```
protected Grim_Reaper is
  procedure Last_Gasp(C: Cause_Of_Termination; T: Task_Id; X: Exception_Occurrence);
end Grim_Reaper;
```

We can then nominate Last_Gasp as the protected procedure to be called when task T dies by

```
Set_Specific_Handler(T'Identity, Grim_Reaper.Last_Gasp'Access);
```

The body of the protected procedure Last_Gasp might then output various diagnostic messages

```
procedure Last_Gasp(C: Cause_Of_Termination; T: Task_Id; X: Exception_Occurrence) is
begin
  case C is
    when Normal => null;
    when Abnormal =>
      Put("Something nasty happened"); ...
    when Unhandled_Exception =>
      Put("Unhandled exception occurred"); ...
```

```
   end case;
end Last_Gasp;
```

There are three possible reasons for termination, it could be normal, abnormal, or caused by an unhandled exception. In the last case the parameter X gives details of the exception occurrence.

Another area of increased flexibility in Ada 2005 is that of task dispatching policies. In Ada 95, the only predefined policy is FIFO_Within_Priorities although other policies are permitted. Ada 2005 provides further pragmas, policies and packages which facilitate many different mechanisms such as non-preemption within priorities, the familiar Round Robin using timeslicing, and the more recently acclaimed Earliest Deadline First (EDF) policy. Moreover, it is possible to mix different policies according to priority level within a partition.

Various facilities are provided by the package Ada.Dispatching plus two child packages

Ada.Dispatching – this is the root package and simply declares an exception Dispatching_Policy_Error.

Ada.Dispatching.Round_Robin – this enables the setting of the time quanta for time slicing within one or more priority levels.

Ada.Dispatching.EDF – this enables the setting of the deadlines for various tasks.

A policy can be selected for a whole partition by one of

```
pragma Task_Dispatching_Policy(Non_Preemptive_FIFO_Within_Priorities);

pragma Task_Dispatching_Policy(Round_Robin_Within_Priorities);

pragma Task_Dispatching_Policy(EDF_Across_Priorities);
```

In order to mix different policies across different priority levels we use the pragma Priority_Specific_Dispatching with various policy identifiers thus

```
pragma Priority_Specific_Dispatching(Round_Robin_Within_Priorities, 1, 1);
pragma Priority_Specific_Dispatching(EDF_Across_Priorities, 2, 10);
pragma Priority_Specific_Dispatching(FIFO_Within_Priorities, 11, 24);
```

This sets Round Robin at priority level 1, EDF at levels 2 to 10, and FIFO at levels 11 to 24.

The final topic in this section concerns the core language and not the Real-Time Systems annex. Ada 2005 introduces a means whereby object oriented and real-time features can be closely linked together through inheritance.

Recall from Section 3.1 that we can declare an interface to be limited thus

```
type LI is limited interface;
```

We can also declare an interface to be synchronized, task, or protected thus

```
type SI is synchronized interface;
type TI is task interface;
type PI is protected interface;
```

A task interface or protected interface has to be implemented by a task type or protected type respectively. However, a synchronized interface can be implemented by either a task type or a protected type. These interfaces can also be composed with certain restrictions. Detailed examples will be given in a later paper.

3.5 Exceptions, numerics, generics etc

As well as the major features discussed above there are also a number of improvements in various other areas.

There are two small changes concerning exceptions. One is that we can give a message with a raise statement, thus

```
raise Some_Error with "A message";
```

This is a lot neater than having to write (as in Ada 95)

```
Ada.Exceptions.Raise_Exception(Some_Error'Identity, "A message");
```

The other change concerns the detection of a null exception occurrence which might be useful in a package analysing a log of exceptions. The problem is that exception occurrences are of a limited private type and so we cannot compare an occurrence with Null_Occurrence to see if they are equal. In Ada 95 applying the function Exception_Identity to a null occurrence unhelpfully raises Constraint_Error. This has been changed in Ada 2005 to return Null_Id so that we can now write

```
procedure Process_Ex(X: Exception_Occurrence) is
begin
  if Exception_Identity(X) = Null_Id then
    -- process the case of a Null_Occurrence
  ...
end Process_Ex;
```

Ada 95 introduced modular types which are of course unsigned integers. However it has in certain cases proved very difficult to get unsigned integers and signed integers to work together. This is a trivial matter in fragile languages such as C but in Ada the type model has proved obstructive. The basic problem is converting a value of a signed type which happens to be negative to an unsigned type. Thus suppose we want to add a signed offset to an unsigned address value, we might have

```
type Offset_Type is range –(2**31) .. 2**31–1;
type Address_Type is mod 2**32;

Offset: Offset_Type;
Address: Address_Type;
```

We cannot just add Offset to Address because they are of different types. If we convert the Offset to the address type then we might get Constraint_Error and so on. The solution in Ada 2005 is to use a new functional attribute S'Mod which applies to any modular subtype S and converts a universal integer value to the modular type using the corresponding mathematical mod operation. So we can now write

```
Address := Address + Address_Type'Mod(Offset);
```

Another new attribute is Machine_Rounding. This enables high-performance conversions from floating point types to integer types when the exact rounding does not matter.

The third numeric change concerns fixed point types. It was common practice for some Ada 83 programs to define their own multiply and divide operations, perhaps to obtain saturation arithmetic. These programs ran afoul of the Ada 95 rules that introduced universal fixed operations and resulted in ambiguities. Without going into details, this problem has been fixed in Ada 2005 so that user-defined operations can now be used.

Ada 2005 has several new pragmas. The first is

```
pragma Unsuppress(Identifier);
```

where the identifier is that of a check such as Range_Check. The general idea is to ensure that checks are performed in a declarative region irrespective of the use of a corresponding pragma Suppress. Thus we might have a type My_Int that behaves as a saturated type. Writing

```
function "*" (Left, Right: My_Int) return My_Int is
  pragma Unsuppress(Overflow_Check);
begin
  return Integer(Left) * Integer(Right);
exception
  when Constraint_Error =>
    if (Left>0 and Right>0) or (Left<0 and Right<0) then
      return My_Int'Last;
    else
      return My_Int'First;
    end if;
end "*";
```

ensures that the code always works as intended even if checks are suppressed in the program as a whole. Incidentally the On parameter of pragma Suppress which never worked well has been banished to Annex J.

Many implementations of Ada 95 support a pragma Assert and this is now consolidated into Ada 2005. The general idea is that we can write pragmas such as

```
pragma Assert(X >50);
```

```
pragma Assert(not Buffer_Full, "buffer is full");
```

The first parameter is a Boolean expression and the second optional parameter is a string. If at the point of the pragma at execution time, the expression is False then action can be taken. The action is controlled by another pragma Assertion_Policy which can switch the assertion mechanism on and off by one of

```
pragma Assertion_Policy(Check);
```

```
pragma Assertion_Policy(Ignore);
```

If the policy is to check then the exception Assertion_Error is raised with the message, if any. This exception is declared in the predefined package Ada.Assertions. There are some other facilities as well.

The pragma No_Return is also related to exceptions. It can be applied to a procedure (not to a function) and indicates that the procedure never returns normally but only by propagating an exception. Thus we might have

```
procedure Fatal_Error(Message: in String);
pragma No_Return(Fatal_Error);
```

And now whenever we call Fatal_Error the compiler is assured that control is not returned and this might enable some optimization or better diagnostic messages.

Note that this pragma applies to the predefined procedure Ada.Exceptions.Raise_Exception.

Another new pragma is Preelaborable_Initialization. This is used with private types and indicates that the full type will have preelaborable initialization. A number of examples occur with the predefined packages such as

```
pragma Preelaborable_Initialization(Controlled);
```

in Ada.Finalization.

Finally, there is the pragma Unchecked_Union. This is useful for interfacing to programs written in C that use the concept of unions. Unions in C correspond to variant types in Ada but do not store any discriminant which is entirely in the mind of the C programmer. The pragma enables a C union to be mapped to an Ada variant record type by omitting the storage for the discriminant.

If the C program has

```
union {
  double spvalue;
  struct {
    int length;
    double* first;
    } mpvalue;
} number;
```

then this can be mapped in the Ada program by

```
type Number(Kind: Precision) is
  record
    case Kind is
      when Single_Precision =>
        SP_Value: Long_Float;
      when Multiple_Precision =>
        MP_Value_Length: Integer;
        MP_Value_First: access Long_Float;
    end case;
  end record;
pragma Unchecked_Union(Number);
```

One problem with pragmas (and attributes) is that many implementations have added implementation defined ones (as they are indeed permitted to do). However, this can impede portability from one implementation to another. To overcome this there are further Restrictions identifiers so we can write

```
pragma Restrictions(No_Implementation_Pragmas, No_Implementation_Attributes);
```

Observe that one of the goals of Ada 2005 has been to standardize as many of the implementation defined attributes and pragmas as possible.

Readers might care to consider the paradox that GNAT has an (implementation-defined) restrictions identifier No_Implementation_Restrictions.

Another new restrictions identifier prevents us from inadvertently using features in Annex J thus

```
pragma Restrictions(No_Obsolescent_Features);
```

Similarly we can use the restrictions identifier No_Dependence to state that a program does not depend on a given library unit. Thus we might write

```
pragma Restrictions(No_Dependence => Ada.Command_Line);
```

Note that the unit mentioned might be a predefined library unit as in the above example but it can also be used with any library unit.

The final new general feature concerns formal generic package parameters. Ada 95 introduced the ability to have formal packages as parameters of generic units. This greatly reduced the need for long generic parameter lists since the formal package encapsulated them.

Sometimes it is necessary for a generic unit to have two (or more) formal packages. When this happens it is often the case that some of the actual parameters of one formal package must be identical to those of the other. In order to permit this there are two forms of generic parameters. One possibility is

```
generic
  with package P is new Q(<>);
package Gen is ...
```

and then the package Gen can be instantiated with any package that is an instantiation of Q. On the other hand we can have

```
generic
  with package R is new S(P1, P2, ... );
package Gen is ...
```

and then the package Gen can only be instantiated with a package that is an instantiation of S with the given actual parameters P1, P2 etc.

These mechanisms are often used together as in

```
generic
  with package P is new Q(<>);
  with package R is new S(P.F1);
package Gen is ...
```

This ensures that the instantiation of S has the same actual parameter (assumed only one in this example) as the parameter F1 of Q used in the instantiation of Q to create the actual package corresponding to P.

There is an example of this in one of the packages for vectors and matrices in ISO/IEC 13813 which is now incorporated into Ada 2005 (see Section 3.6). The generic package for complex arrays has two package parameters. One is the corresponding package for real arrays and the other is the package Generic_Complex_Types from the existing Numerics annex. Both of these packages have a floating type as their single formal parameter and it is important that both instantiations use the same floating type (eg both Float and not one Float and one Long_Float) otherwise a terrible mess will occur. This is assured by writing (using some abbreviations)

```
with ... ;
generic
  with package Real_Arrays is new Generic_Real_Arrays(<>);
  with package Complex_Types is new Generic_Complex_Types(Real_Arrays.Real);
package Generic_Complex_Arrays is ...
```

Well this works fine in simple cases (the reader may wonder whether this example is simple anyway) but in more elaborate situations it is a pain. The trouble is that we have to give all the parameters for the formal package or none at all in Ada 95.

Ada 2005 permits only some of the parameters to be specified, and any not specified can be indicated using the box. So we can write any of

```
with package Q is new R(P1, P2, F3 => <>);
with package Q is new R(P1, others => <>);
with package Q is new R(F1 => <>, F2 => P2, F3 => P3);
```

Note that the existing form (<>) is now deemed to be a shorthand for (**others** => <>). As with aggregates, the form <> is only permitted with named notation.

Examples using this new facility will be given in a later paper.

3.6 Standard library

There are significant improvements to the standard library in Ada 2005. One of the strengths of Java is the huge library that comes with it. Ada has tended to take the esoteric view that it is a language for constructing programs from components and has in the past rather assumed that the components would spring up by magic from the user community. There has also perhaps been a reluctance to specify standard components in case that preempted the development of better ones. However, it is now recognized that standardizing useful stuff is a good thing. And moreover, secondary ISO standards are not very helpful because they are almost invisible. Ada 95 added quite a lot to the predefined library and Ada 2005 adds more.

First, there are packages for manipulating vectors and matrices already mentioned in Section 3.5 when discussing formal package parameters. There are two packages, Ada.Numerics.Generic_Real_Arrays for real vectors and matrices and Ada.Numerics.Generic_Complex_Arrays for complex vectors and matrices. They can be instantiated according to the underlying floating point type used. There are also nongeneric versions as usual.

These packages export types for declaring vectors and matrices and many operations for manipulating them. Thus if we have an expression in mathematical notation such as

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{z}$$

where $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$ are vectors and $\boldsymbol{A}$ is a square matrix, then this calculation can be simply programmed as

```
X, Y, Z: Real_Vector(1 .. N);
A: Real_Matrix(1 .. N, 1 .. N);
...
Y := A * X + Z;
```

and the appropriate operations will be invoked. The packages also include subprograms for the most useful linear algebra computations, namely, the solution of linear equations, matrix inversion and determinant evaluation, plus the determination of eigenvalues and eigenvectors for symmetric matrices (Hermitian in the complex case). Thus to determine X given Y, Z and A in the above example we can write

```
X := Solve(A, Y – Z);
```

It should not be thought that these Ada packages in any way compete with the very comprehensive BLAS (Basic Linear Algebra Subprograms). The purpose of the Ada packages is to provide simple implementations of very commonly used algorithms (perhaps for small embedded systems or for prototyping) and to provide a solid framework for developing bindings to the BLAS for more demanding situations. Incidentally, they are in the Numerics annex.

Another (but very trivial) change to the Numerics annex is that nongeneric versions of Ada.Text_IO.Complex_IO have been added in line with the standard principle of providing nongeneric versions of generic predefined packages for convenience. Their omission from Ada 95 was an oversight.

There is a new predefined package in Annex A for accessing tree-structured file systems. The scope is perhaps indicated by this fragment of its specification

```
with ...
package Ada.Directories is
  -- Directory and file operations
  function Current_Directory return String;
  procedure Set_Directory(Directory: in String);
  ...
```

```
    -- File and directory name operations
    function Full_Name(Name: in String) return String;
    function Simple_Name(Name: in String) return String;
    ...
    -- File and directory queries
    type File_Kind is (Directory, Ordinary_File, Special_File);
    type File_Size is range 0 .. implementation-defined;
    function Exists(Name: in String) return Boolean;
    ...
    -- Directory searching
    type Directory_Entry_Type is limited private;
    type Filter_Type is array (File_Kind) of Boolean;
    ...
    -- Operations on directory entries
    ...
end Ada.Directories;
```

The package contains facilities which will be useful on any Unix or Windows system. However, it has to be recognized that like Ada.Command_Line it might not be supportable on every environment.

There is also a package Ada.Environment_Variables for accessing the environment variables that occur in most operating systems.

A number of additional subprograms have been added to the existing string handling packages. There are several problems with the Ada 95 packages. One is that conversion between bounded and unbounded strings and the raw type String is required rather a lot and is both ugly and inefficient. For example, searching only part of a bounded or unbounded string can only be done by converting it to a String and then searching the appropriate slice (or by making a truncated copy first).

In brief the additional subprograms are as follows

_ Three further versions of function Index with an additional parameter From indicating the start of the search are added to each of Strings.Fixed, Strings.Bounded and Strings.Unbounded.

_ A further version of function Index_Non_Blank is similarly added to all three packages.

_ A procedure Set_Bounded_String with similar behaviour to the function To_Bounded_String is added to Strings.Bounded. This avoids the overhead of using a function. A similar procedure Set_Unbounded_String is added to Strings.Unbounded.

_ A function and procedure Bounded_Slice are added to Strings.Bounded. These avoid conversions from type String. A similar function and procedure Unbounded_Slice are added to Strings.Unbounded.

As well as these additions there is a new package Ada.Text_IO.Unbounded_IO for the input and output of unbounded strings. This again avoids unnecessary conversion to the type String. Similarly, there is a generic package Ada.Text_IO.Bounded_IO; this is generic because the package Strings.Bounded has an inner generic package which is parameterized by the maximum string length.

Finally, two functions Get_Line are added to Ada.Text_IO itself. These avoid difficulties with the length of the string which occurs with the existing procedures Get_Line.

In Ada 83, program identifiers used the 7-bit ASCII set. In Ada 95 this was extended to the 8-bit Latin-1 set. In Ada 2005 this is extended yet again to the entire ISO/IEC 10646:2003 character repertoire. This means that identifiers can now use Cyrillic and Greek characters. Thus we could extend the animal example by

```
_т____: access Pig renames Napoleon;
Πεγασυς: Horse;
```

In order to encourage us to write our mathematical programs nicely the additional constant

```
π: constant := Pi;
```

has been added to the package Ada.Numerics in Ada 2005.

In a similar way types Wide_String and Wide_Character were added to Ada 95. In Ada 2005 this process is also extended and a set of wide-wide types and packages for 32-bit characters are added. Thus we have types Wide_Wide_Character and Wide_Wide_String and so on.

A major addition to the predefined library is the package Ada.Containers and its children plus some auxiliary child functions of Ada.Strings. These are very important and considerable additions to the predefined capability of Ada and bring the best in standard data structure manipulation to the fingers of every Ada programmer. The scope is perhaps best illustrated by listing the units involved.

Ada.Containers – this is the root package and just declares types Hash_Type and Count_Type which are an implementation-defined modular and integer type respectively.

Ada.Strings.Hash – this function hashes a string into the type Hash_Type. There are also versions for bounded and unbounded strrings.

Ada.Containers.Vectors – this is a generic package with parameters giving the index type and element type of a vector plus "=" for the element type. This package declares types and operations for manipulating vectors. (These are vectors in the sense of flexible arrays and not the mathematical vectors used for linear algebra as in the vectors and matrices packages mentioned earlier.) As well as subprograms for adding, moving and removing elements there are also generic subprograms for searching, sorting and iterating over vectors.

Ada.Containers.Doubly_Linked_Lists – this is a generic package with parameters giving the element type and "=" for the element type. This package declares types and operations for manipulating doubly-linked lists. It has similar functionality to the vectors package. Thus, as well as subprograms for adding, moving and removing elements there are also generic subprograms for searching, sorting and iterating over lists.

Ada.Containers.Hashed_Maps – this is a generic package with parameters giving a key type and an element type plus a hash function for the key, a function to test for equality between keys and "=" for the element type. It declares types and operations for manipulating hashed maps.

Ada.Containers.Ordered_Maps – this is a similar generic package for ordered maps with parameters giving a key type and an element type and "<" for the key type and "=" for the element type.

Ada.Containers.Hashed_Sets – this is a generic package with parameters giving the element type plus a hash function for the elements and a function to test for equality between elements. It declares types and operations for manipulating hashed sets.

Ada.Containers.Ordered_Sets – this is a similar generic package for ordered sets with parameters giving the element type and "<" and "=" for the element type.

There are then another six packages with similar functionality but for indefinite types with corresponding names such as Ada.Containers.Indefinite_Vectors.

Ada.Containers.Generic_Array_Sort – this is a generic procedure for sorting arrays. The generic parameters give the index type, the element type, the array type and "<" for the element type. The array type is unconstrained.

Finally there is a very similar generic procedure Ada.Containers.Generic_Constrained_Array_Sort but for constrained array types.

It is hoped that the above list gives a flavour of the capability of the package Containers. Some examples of the use of the facilities will be given in a later paper.

Finally, there are further packages for manipulating times (that is of type Ada.Calendar.Time and not Ada.Real_Time.Time and thus more appropriate in a discussion of the predefined library than the real-time features). The package Ada.Calendar has a number of obvious omissions and in order to rectify this the following packages are added.

Ada.Calendar.Time_Zones – this declares a type Time_Offset describing in minutes the difference between two time zones and a function UTC_Time_Offset which given a time returns the difference between the time zone of Calendar at that time and UTC (Coordinated Universal Time which is close to Greenwich Mean Time). It also has an exception which is raised if the time zone of Calendar is not known (maybe the clock is broken).

Ada.Calendar.Arithmetic – this declares various types and operations for coping with leap seconds.

Ada.Calendar.Formatting – this declares further types and operations for dealing with formatting and related matters.

Most of the new calendar features are clearly only for the chronological addict but the need for them does illustrate that this is a tricky area. However, a feature that all will appreciate is that the package Ada.Calendar.Formatting includes the following declarations

```
type Day_Name is (Monday, Tuesday, Wednesday,
                  Thursday, Friday, Saturday, Sunday);

function Day_Of_Week(Date: Time) return Day_Name;
```

There is also a small change in the parent package Ada.Calendar itself. The subtype Year_Number is now

```
subtype Year_Number is Integer range 1901 .. 2399;
```

This reveals confidence in the future of Ada by adding another three hundred years to the range of dates.

4 Conclusions

This overview of Ada 2005 should have given the reader an appreciation of the important new features in Ada 2005. Some quite promising features failed to be included partly because the need for them was not clear and also because a conclusive design proved elusive. We might think of them as Forthcoming Attractions for any further revision!

Some esoteric topics have been omitted in this overview; they concern features such as: streams, object factory functions, the partition control system in distributed systems, partition elaboration policy for high integrity systems, a subtlety regarding overload resolution, the title of Annex H, quirks of access subtypes, rules for pragma Pure, and the classification of various units as pure or preelaborable.

Further papers will expand on the six major topics of this overview in more detail.

It is worth briefly reviewing the guidelines (see Section 2 above) to see whether Ada 2005 meets them. Certainly the Ravenscar profile has been added and the problem of mutually dependent types across packages has been solved.

The group A items were about real-time and high-integrity, static error checking and interfacing. Clearly there are major improvements in the real-time area. And high-integrity and static error checking are addressed by features such as the **overriding** prefix, various pragmas such as Unsuppress and Assert and additional Restrictions identifiers. Better interfacing is provided by the pragma Unchecked_Union and the Mod attribute.

The group B items were about improvements to the OO model, the need for a Java-like interface feature and better interfacing to other OO languages. Major improvements to the OO model are brought by the prefixed (Obj.Op) notation and more flexible access types. The Java-like interface feature has been added and this provides better interfacing.

The final direct instruction was to incorporate the vectors and matrices stuff and this has been done. There are also many other improvements to the predefined library as we have seen.

It seems clear from this brief check that indeed Ada 2005 does meet the objectives set for it.

Finally, I need to thank all those who have helped in the preparation of this paper. First I must acknowledge the financial support of Ada-Europe and the Ada Resource Association. And then I must thank those who reviewed earlier versions. There are almost too many to name, but I must give special thanks to Randy Brukardt, Pascal Leroy and Tucker Taft of the ARG, to my colleagues on the UK Ada Panel (BSI/IST/5/–/9), and to James Moore of WG9. I am especially grateful for a brilliant suggestion of Randy Brukardt which must be preserved for the pleasure of future generations. He suggests that this document when complete be called the Ada Language Enhancement Guide. This means that if combined with the final Ada Reference Manual, the whole document can then be referred to as the ARM and ALEG. Thanks Randy.

References

[1] ISO/IEC JTC1/SC22/WG9 N412 (2002) *Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of the Amendment.*

[2] ISO/IEC 8652:1995/COR 1:2001, *Ada Reference Manual – Technical Corrigendum 1.*

[3] S. T. Taft et al (eds) (2001) *Consolidated Ada Reference Manual*, LNCS 2219, Springer-Verlag.

[4] ISO/IEC TR 24718:2004 (2004) *Guide for the use of the Ada Ravenscar Profile in high integrity systems.* This is based on University of York Technical Report YCS-2003-348 (2003).

[5] ISO/IEC 13813:1997 (1997) *Generic packages of real and complex type declarations and basic operations for Ada (including vector and matrix types).*

[6] J. G. P. Barnes (1998) *Programming in Ada 95*, 2nd ed., Addison-Wesley.