7 Packages

There are a number of important changes to the language addressed in this chapter. Many of these are associated with tagged types such as the addition of private extensions. Another important change is the introduction of controlled types which are implemented using tagged types. In summary the changes are

- A private type may be marked as tagged. The rules regarding private types and discriminants are considerably relaxed.

- Private extensions are introduced in order to allow type extension of tagged types where the extension part is private.

- A deferred constant can be of any type, not just a private type.

- The property of being limited is now separated from that of being private. They are both view properties.

- Controlled types are added. These permit finalization and the user definition of assignment.

An important change regarding library packages is that they can have a body only when one is required; this is discussed in 10.4.

7.1 Private Types and Extensions

To allow for the extension of private types, the modifier tagged may be specified in a private type declaration. A tagged private type must be implemented with a tagged record type, or by deriving from some other tagged type.

We considered allowing a tagged private type to be derived from an untagged type. However, this added potential implementation complexity because the parent type might not have a layout optimized for referencing components added in later extensions. There is a simple work-around; the tagged private type can be implemented as a tagged record type with a single component of the desired parent type.

In Ada 95, we consider a private type to be a composite type outside the scope of its full type declaration. This is primarily a matter of presentation.

Although a private tagged type must be implemented as a tagged type, the reverse is not the case. A private untagged type can be implemented as a tagged or untagged type. In other words if the partial view is tagged then the full view must be tagged; if the partial view is untagged then the full view may or may not be tagged. We thus have to be careful when saying that a type is tagged, strictly we should talk about the view being tagged. The relationship between the full and partial view of a type is discussed further in 7.3 when we also consider limited types.

A good example of the use of a tagged type to implement a private type which is not tagged is given by the type `Unbounded_String` in the package `Ada.Strings.Unbounded`. The sample implementation discussed in A.2.6 shows how `Unbounded_String` is controlled and thus derived from `Finalization.Controlled`.

We have generalized discriminants so that a derived type declaration may include a discriminant part. Therefore, it is now permissible for a private type with discriminants to be implemented by a derived type declaration. This was not possible in Ada 83.

In addition, a discriminant part may be provided in a task or protected type declaration. As a consequence, a limited private type with discriminants may be implemented as a task or protected type.

Another improvement mentioned in II.12 is that a private type without discriminants may now be implemented as any definite type including a discriminated record type with defaulted discriminants.

A tagged (parent) type may be extended with a private extension part. This allows one type to visibly extend another, while keeping the names and types of the components in the extension part private to the package where the extension is defined.

For a private extension in the visible part, a corresponding record extension must be defined in the private part. Note that the record extension part may be simply

```
with null record
```

which uses an abbreviated form of syntax for null records. This abbreviated form was introduced precisely because null extensions are fairly common.

An extension aggregate is only allowed where the full view of the extended type is visible even if the extension part is null. Note that a private extension may always be (view) converted to its parent type. See also 3.6.1 for the case where the ancestor part is abstract.

The interplay between type extension and visibility is illustrated by the following somewhat contrived example

```
package P1 is
   type T is tagged
      record
         A: Type_A;
      end record;
   type T1 is new T with private;
private
   type T1 is new T with
      record
         B: Type_B;
      end record;
end P1;

with P1; use P1;
package P2 is
   type T2 is new T1 with
      record
         C: Type_C;
      end record;
end P2;

with P2; use T2
package body P1 is

   X1: T1;      -- can write X1.B
   X2: T2;      -- cannot write X2.B
   XX: T1 renames T1(X2);
   ...
   XX.B :=      -- changes B component of X2

end P1;
```

The type `T` has a component `A`. The type `T1` is extended from `T` with the additional component `B` but the extension is not visible outside the private part and body of `P1`. The type `T2` is in turn extended from `T1` with a further component `C`. However, although `T2` has a component `B` it is not visible for any view of `T2`, since the declaration of `T2` sees only the partial view of its parent `T1`. So the `B` component of `T2` is not visible in the package body `P1` even though that component of its parent is indeed visible from there. But of course we can do a view conversion of an object of the type `T2` and then obtain access to the component `B` provided we are at a place that has the full view of `T1`.

It is important that the invisible `B` component of `T2` be invisible for all views since otherwise we would run into a problem if the additional component of `T2` were also called `B` (this is allowed because the existing `B` component is not visible at the point where the additional component is declared and the potential clash could not be known to the writer of `P2`). But if there were a view such that all components of `T2` were visible (such as perhaps might be expected in the body of `P1`) then `X2.B` would be ambiguous.

The important general principle is that the view we get of a type is the view relating to the declaration of it that is visible to us rather than simply where we are looking from.

7.1.1 Private Operations

A tagged type may have primitive operations declared in a private part. These operations will then not be available to all views of the type although nevertheless they will always be in the dispatch table. We noted in 3.6.2 that an abstract type can never have private abstract operations.

A private operation can be overridden even if the overriding operation actually occurs at a place where the private operation is not visible. Consider

```
package P is
   type A is tagged ...;
private
   procedure Op(X: A);
end P;

package P.Child is
   type B is new A with ...;
   procedure Op(X: B);
private
   -- the old Op would have been visible here
end P.Child;
```

The type `A` has a private dispatching operation `Op`. The type `B` is an extension of `A` and declares an operation `Op`. This overrides the private inherited operation of the same name despite the fact that the private operation is not visible at the point of the declaration of the new operation. But the key point is that within the private part of the child package, the old `Op` would have become visible and this is still within the region where it is possible to add primitive operations for `B`. It is not possible for both operations to be primitive and visible at the same place and it would be impossible for them to share the same slot in the dispatch table. Accordingly the new operation overrides the old. Moreover, they must conform. For a practical example see 13.5.1.

On the other hand if the new operation is declared at a point such that the visibility of the two never clash in this way such as in the following

```
package P is
   type A is tagged ...;
   procedure Curious(X: A'Class);
private
   procedure Op(X: A);  -- first one
end P;
```

```
with P; use P;
package Outside is
   type B is new A with ...;
   procedure Op(X: B);  -- second one
end Outside;
```

then the two operations do not clash and occupy different slots in the dispatch table. Moreover they need not conform since they are quite independent. So in fact B does actually have both operations; it inherits the private one from A and has its own different one. We will dispatch to one or the other according to how a dispatching call is made. The first one is of course a dispatching operation of A'Class whereas the second is a dispatching operation of B'Class. The procedure Curious might be

```
procedure Curious(X: A'Class) is
begin
   Op(X);   -- dispatch to first Op
end Curious;
```

and then

```
Object: B;
...
Curious(Object);
```

will call the inherited hidden operation of B which will apply itself to the part of B inherited from A. This hidden operation is of course just that inherited from A; it cannot be changed and hence can know nothing of the additional components of B.

Note further that we could declare a further type extension from B at a place where the operation of A is also visible. This could be in the private part of a child of P or in a package inner to the body of P. For example

```
with Outside; use Outside;
package P.Child is

private
   type C is new B with ...;

end P.Child;
```

In such a case C inherits both operations from B in the sense that they both occupy slots in the dispatch table. But again the operation acquired indirectly from A is totally invisible; it does not matter that the operation of A is visible at this point; all that matters is that C cannot see the corresponding operation of B. This is another example of the principle mentioned at the end of the previous section that the view we get of a type is the view relating to the declaration of it that is visible to us rather than where we are looking from; or in other words the history of how B got its operations is irrelevant.

7.2 Deferred Constants

In Ada 83, deferred constants were only permitted in the visible part of a package and only if their type was private and was declared in the same visible part [RM83 7.4(4)].

In Ada 95, this restriction is relaxed, so that a deferred constant of any type may be declared immediately within the visible part of a package, provided that the full constant declaration is given in the private part of the package. This eliminates the anomaly that prevented a constant of a composite type with a component of a private type from being declared, if the composite type was declared in the same visible part as the private type.

Another advantage of deferred constants is that in some cases, the initial value depends on attributes of objects or types that are declared in the private part. For example, one might want to export an access to constant value designating a variable in the private part. This prevents the external user from changing the value although the package of course can change the value. This is another example of having two different views of an entity; in this case a constant view and a variable view.

```
      type Int_Ptr is access constant Integer;
      The_Ptr: constant Int_Ptr;  -- deferred constant
   private
      Actual_Int: aliased Integer;
       -- is a variable so we do not need an initial value
       The_Ptr: constant Int_Ptr := Actual_Int'Access;
       -- full definition of deferred constant
```

Note that a deferred constant can also be completed by an `Import` pragma thereby indicating that the object is external to the Ada program. See Part Three for details.

A small point regarding deferred constants is that they no longer have to be declared with just a subtype mark; a subtype indication including an explicit constraint is allowed. Such a constraint must statically match that in the full constant declaration.

7.3 Limited Types

As in Ada 83, a limited type is one for which assignment is not allowed (user-defined assignment is not allowed either, see 7.4). However, the property of being limited is no longer tied to private types. Any record type can be declared as limited by the inclusion of **limited** in its definition. Thus the type `Outer` in 6.4.3 is limited. Task and protected types are also limited and a type is limited if it has any components that are limited. Only a limited type can have access discriminants. Finally, a derived type is limited if its parent is limited.

Limited is a sort of view property in that whether a type is limited or not may depend upon from where it is viewed. This is obvious with a limited private type where the full view might not be limited. However, it can occur even in the nonprivate case. Consider

```
   package P is
      type T is limited private;
      type A is array (...) of T;
   private
      type T is new Integer;
      -- at this point A becomes nonlimited
   end P;
```

where the type `A` starts off being limited because its components are limited. However, after the full declaration of `T`, its components are no longer limited and so `A` becomes nonlimited.

Note that in the case of a tagged type, it must have **limited** in its definition (or that of its ancestor) if it has limited components. This prevents a tagged type from the phenomenon of becoming nonlimited. Otherwise one might extend from a limited view with a limited component (such as a task) and then in the full view try to do assignment as in the following variation of the previous example.

```
   package P is
      type T is limited private;
      type R is tagged  -- illegal, must have explicit limited
```

```
      record
         X: T;
      end record;
private
   type T is new Integer;
   -- at this point R would become nonlimited
end P;

package Q is
   type T2 is new R with
      record
         A: Some_Task;
      end record;
end Q;
```

The problem is that the body of `P` would see a nonlimited view of `T` and hence assignment would be defined for `T'Class` and so it would be possible to do an assignment on the type `T2` by a dispatching operation in the body of `P`.

So, in the case of a tagged private type (that is a type for which both partial and full views are tagged), both partial and full views must be limited or not together; it is not possible for the partial view to be limited and the full view not to be limited. On the other hand if the partial view is untagged and limited then the full view can be any combination including tagged and nonlimited. The various possibilities are illustrated in Table 7-1; only those combinations marked as OK are legal.

full view	partial view untagged limited	partial view untagged nonlimited	partial view tagged limited	partial view tagged nonlimited
untagged limited	OK			
untagged nonlimited	OK	OK		
tagged limited	OK		OK	
tagged nonlimited	OK	OK		OK

Table 7-1: Full and Partial Views

A consequence of the rules is that, in the case of type extension, if the parent type is not limited, then the extension part cannot have any limited components. (Note that the rules regarding actual and formal generic parameters are somewhat different; the actual type corresponding to a formal limited tagged type does not have to be limited. This is because type extension is not permitted in the generic body.)

There was a pathological situation in Ada 83 whereby a function could return a local task (all one could do with it outside was apply the attributes `Terminated` and `Callable`); this was a nuisance because all the storage for the local task could not be properly relinquished on the return.

In Ada 95 there is an accessibility check that prevents such difficulties. In essence we are not allowed to return a local object of a limited type (there are some subtle exceptions for which see [RM95 6.5]).

An important consequence of a function result being treated as an object is that it can be renamed. This means that we can "remember" the result of a function even in the case of a limited type. For example, the function `Text_IO.Current_Output` returns the current default output file. In Ada 83 it was difficult to remember this and then reset the default value back after having used some other file as current output in the meantime; it could be done but only with a contorted use of parameters. In Ada 95 we can write

```
Old_File: File_Type renames Current_Output;
...
Set_Output(Old_File);
```

and the renaming holds onto the object which behaves much as an **in** parameter. But see also Part Three for a more general solution to the problem of remembering a current file.

7.4 Controlled Types

To preserve abstraction, while providing automatic reclamation of resources, Ada 95 provides controlled types that have initialization and finalization operations associated with them. A number of different approaches were considered and rejected during the evolution of Ada 95. The final solution has the merit of allowing user-defined assignment and also solves the problem of returning limited types mentioned in the previous section.

The general principle is that there are three distinct primitive activities concerning the control of objects

- initialization after creation,
- finalization before destruction (includes overwriting),
- adjustment after assignment.

and the user is given the ability to provide appropriate procedures which are called to perform whatever is necessary at various points in the life of an object. These procedures are `Initialize`, `Finalize` and `Adjust` and take the object as an **in out** parameter.

To see how this works, consider

```
declare
   A: T;          -- create A, Initialize(A)
begin
   A := E;        -- Finalize(A), copy value, Adjust(A)
   ...
end;              -- Finalize(A)
```

After `A` is declared and any normal default initialization carried out, the `Initialize` procedure is called. On the assignment, `Finalize` is first called to tidy up the old object about to be overwritten and thus destroyed, the physical copy is then made and finally `Adjust` is called to do whatever might be required for the new copy. At the end of the block `Finalize` is called once more before the object is destroyed. Note, of course, that the user does not have to physically write the calls of the three control procedures, they are called automatically by the compiled code.

In the case of a nested structure where inner components might themselves be controlled, the rules are that components are initialized and adjusted before the object as a whole and on finalization everything is done in the reverse order.

There are many other situations where the control procedures are invoked such as when calling allocators, evaluating aggregates and so on; for details the reader should consult [RM95].

In order for a type to be controlled it has to be derived from one of two tagged types declared in the library package `Ada.Finalization` whose specification is given in [RM95 7.6] and which is repeated here for convenience

```
package Ada.Finalization is

   pragma Preelaborate(Finalization);

   type Controlled is abstract tagged private;

   procedure Initialize(Object: in out Controlled);
   procedure Adjust(Object: in out Controlled);
   procedure Finalize(Object: in out Controlled);

   type Limited_Controlled is abstract tagged limited private;

   procedure Initialize(Object: in out Limited_Controlled);
   procedure Finalize(Object: in out Limited_Controlled);

private
   ...
end Ada.Finalization;
```

There are distinct abstract types for nonlimited and limited types. Naturally enough the `Adjust` procedure does not exist in the case of limited types because they cannot be copied.

Although the types `Controlled` and `Limited_Controlled` are abstract, nevertheless the procedures `Initialize`, `Adjust` and `Finalize` are not abstract. However they all do nothing which will often prove to be appropriate.

A typical declaration of a controlled type might take the form

```
type T is new Controlled with ...
```

and the user would then provide new versions of the controlling procedures as required. Note incidentally that the form of an extension aggregate mentioned in 3.6.1 where the ancestor part is a subtype name is useful for controlled types since we can write

```
X: T := (Controlled with ...);
```

whereas we cannot use an expression as the ancestor part because its type is abstract.

The capabilities provided take a building block approach and give the programmer fine control of resources. In particular they allow the implementor of an abstraction to ensure that proper cleanup is performed prior to the object becoming permanently inaccessible.

The ability to associate automatic finalization actions with an abstraction is extremely important for Ada, given the orientation toward information hiding, coupled with the many ways that a scope may be exited in Ada (exception, exit, return, abort, asynchronous transfer of control, etc). In many cases, the need for finalization is more of a safety or reliability issue than a part of the visible specification of an abstraction. Most users of an abstraction should not need to know whether the abstraction uses finalization.

A related observation concerns the use of controlled types as generic parameters. We can write a package which adds some properties to an arbitrary controlled type in the manner outlined in 4.6.2. Typically we will call the `Finalize` of the parent as part of the `Finalize` operation of the new type. Consider

```
generic
   type Parent is abstract new Limited_Controlled with private;
```

```
package P is
   type T is new Parent with private;
   ...
private
   type T is new Parent with
      record
         -- additional components
      end record;
   procedure Finalize(Object: in out T);
end P;

package body P is
   ...
   procedure Finalize(Object: in out T) is
   begin
      ...  -- operations to finalize the additional components
      Finalize(Parent(Object));  -- finalize the parent
   end Finalize;
end P;
```

This will always work even if the implementation of the actual type corresponding to `Parent` has no overriding `Finalize` itself since the inherited null version from `Limited_Controlled` will then be harmlessly called. See also 12.5.

The approach we have adopted enables the implementation difficulties presented by, for example, exceptions to be overcome. Thus suppose an exception is raised in the middle of a sequence of declarations of controlled objects. Only those already elaborated will need to be finalized and some mechanism is required in order to record this. Such a mechanism can conveniently be implemented using links which are components of the private types `Controlled` and `Limited_Controlled`; these components are of course quite invisible to the user. Incidentally, this illustrates that an abstract type need not be null.

The following example is of a generic package that defines a sparse array type. The array is extended automatically as new components are assigned into it. `Empty_Value` is returned on dereferencing a never-assigned component. On scope exit, automatic finalization calls `Finalize`, which reclaims any storage allocated to the sparse array.

```
with Ada.Finalization; use Ada;
generic
   type Index_Type is (<>);
     -- index type must be some discrete type
   type Element_Type is private;
     -- component type for sparse array
   Empty_Value : in Element_Type;
     -- value to return for never-assigned components
package Sparse_Array_Pkg is
   -- this generic package defines a sparse array type
   type Sparse_Array_Type is
       new Finalization.Limited_Controlled with private;

   procedure Set_Element(Arr: in out Sparse_Array_Type;
                         Index: in Index_Type;
                         Value: in Element_Type);
     -- set value of an element
     -- extend array as necessary

   function Element(Arr: Sparse_Array_Type;
                    Index: Index_Type) return Element_Type;
     -- return element of array
```

```
         -- return Empty_Value if never assigned

      procedure Finalize(Arr: in out Sparse_Array_Type);
         -- reset array to be completely empty

      -- use default Initialize implementation (no action)
      -- no Adjust for limited types

   private
      -- implement using a B-tree-like representation
      type Array_Chunk;
         -- type completed in package body
      type Array_Chunk_Ptr is access Array_Chunk;
      type Sparse_Array_Type is
         new Finalization.Limited_Controlled with
         record
            Number_Of_Levels: Natural := 0;
            Root_Chunk      : Array_Chunk_Ptr;
         end record;
   end Sparse_Array_Pkg;

   package body Sparse_Array_Pkg is
      type Array_Chunk is ...
         -- complete the incomplete type definition
      procedure Set_Element( ...

      function Element(Arr: Sparse_Array_Type;
                       Index: Index_Type) return Element_Type is
      begin
         if Arr.Root_Chunk = null then
            -- entire array is empty
            return Empty_Value;
         else
            -- look up to see if Index appears
            -- in array somewhere
            ...
         end if;
      end Element;

      procedure Finalize(Arr: in out Sparse_Array_Type) is
      begin
         if Arr.Root_Chunk /= null then
            -- release all chunks of array
            ...
            -- reinitialize array back to initial state
            Arr.Root_Chunk := null;
            Arr.Number_Of_Levels := 0;
         end if;
      end Finalize;
   end Sparse_Array_Pkg;
```

Since the sparse array type is derived from a library level tagged type (`Ada.Finalization.Limited_Controlled`), the generic unit must also be instantiated at the library level. However, an object of the sparse array type defined in the instantiation may be declared in any scope. When that scope is exited, for whatever reason, the storage dynamically allocated to the array will be reclaimed, via an implicit call on `Sparse_Array_Type.Finalize`.

Such a sparse array type may safely be used by subprograms of a long-running program, without concern for progressive loss of storage. When such subprograms return, the storage will always be reclaimed, whether completed by an exception, return, abort, or asynchronous transfer of control.

Our next example illustrates user-defined assignment. Incidentally, it should be noted that many of the cases where user-defined assignment was felt to be necessary in Ada 83 no longer apply because the ability to redefine `"="` has been generalized. Many Ada 83 applications using limited types did so in order to redefine `"="` and as a consequence lost predefined assignment. Their need for user-defined assignment was simply to get back the predefined assignment.

An instance where user-defined assignment would be appropriate occurs in the implementation of abstract sets using linked lists where a deep copy is required as mentioned in 4.4.3.

The general idea is that the set is implemented as a record containing one inner component which is controlled; this controlled component is an access to a linked list containing the various elements. Whenever the controlled component is assigned it makes a new copy of the linked list. Note that the type `Linked_Set` as a whole cannot be controlled because it is derived directly from `Abstract_Sets.Set`. However, assigning a value of the `Linked_Set` causes the inner component to be assigned and then invokes the procedure `Adjust` on the inner component. The implementation might be as follows

```
with Abstract_Sets;
with Ada.Finalization; use Ada.Finalization;
package Linked_Sets is

   type Linked_Set is new Abstract_Sets.Set with private;

   ...  -- the various operations on Linked_Set

private

   type Node;
   type Ptr is access Node;
   type Node is
      record
         Element: Set_Element;
         Next: Ptr;
      end record;

   function Copy(P: Ptr) return Ptr;  -- deep copy

   type Inner is new Controlled with
      record
         The_Set: Ptr;
      end record;

   procedure Adjust(Object: in out Inner);

   type Linked_Set is new Abstract_Sets.Set with
      record
         Component: Inner;
      end record;

end Linked_Sets;

package body Linked_Sets is

   function Copy(P: Ptr) return Ptr is
```

```
      begin
         if P = null then
            return null;
         else
            return new Node'(P.Element, Copy(P.Next));
         end if;
      end Copy;

      procedure Adjust(Object: in out Inner) is
      begin
         Object.The_Set := Copy(Object.The_Set);
      end Adjust;

      ...

   end Linked_Sets;
```

The types `Node` and `Ptr` form the usual linked list containing the elements; `Node` is of course not tagged or controlled. The function `Copy` performs a deep copy of the list passed as parameter. The type `Inner` is controlled and contains a single component referring to the linked list. The procedure `Adjust` for `Inner` performs a deep copy on this single component. The visible type `Linked_Set` is then simply a record containing a component of the controlled type `Inner`. As mentioned above, performing an assignment on the type `Linked_Set` causes `Adjust` to be called on its inner component thereby making the deep copy. But none of this is visible to the user of the package `Linked_Sets`. Observe that we do not need to provide a procedure `Initialize` and that we have not bothered to provide `Finalize` although it would be appropriate to do so in order to discard the unused space.

Finally, we show a canonical example of the use of initialization and finalization and access discriminants for the completely safe control of resources. Consider the following:

```
   type Handle(Resource: access Some_Thing) is
            new Finalization.Limited_Controlled with null record;

   procedure Initialize(H: in out Handle) is
   begin
      Lock(H.Resource);
   end Initialize;

   procedure Finalize(H: in out Handle) is
   begin
      Unlock(H.Resource);
   end Finalize;

   ...

   procedure P(T: access Some_Thing) is
      H: Handle(T);
   begin
      ... -- process T safely
   end P;
```

The declaration of `H` inside the procedure `P` causes `Initialize` to be called which in turn calls `Lock` with the object referred to by the handle as parameter. The general idea is that since we know that `Finalize` will be called no matter how we leave the procedure `P` (including because of an exception or abort) then we will be assured that the unlock operation will always be done. This is a useful technique for ensuring that typical pairs of operations are performed correctly such as opening and closing files. Note that we have to declare the handle locally because that is where the locking is required and hence an access discriminant has to be used in order to avoid accessibility problems. We have to have a handle in the first place so that its declaration is tied to the vital finalization.

Some examples of the use of finalization with asynchronous transfer of control will be found in 9.4.

7.5 Requirements Summary

The major study topic

S4.2-A(2) — Preservation of Abstraction

is directly addressed and satisfied by the introduction of controlled types as discussed in 7.4.

8 Visibility Rules

This is an area of the language which is largely ignored by the normal programmer except when it produces surprising or frustrating consequences. The changes have thus been directed largely towards making the rules clear and consistent and with more helpful consequences. The changes are

- The visibility and scope rules are rewritten to make them consistent and clearer. They also incorporate the consequences of the introduction of the hierarchical library.
- The use type clause is introduced for operators.
- Renaming is now allowed for subprogram bodies, generic units and library units.
- There are also a number of minor improvements such as the preference rules for overload resolution.

8.1 Scope and Visibility

This is an area where there is no substitute for a precise description of the rules. Suffice it to say that the rules for Ada 83 were obscure and probably not truly understood by anybody; a consequence was subtle variation between compilers to the detriment of portability. We will not attempt to summarize the 95 rules but refer the reader to [RM95] for the details.

One important change is that character literals are now treated like other literals with regard to visibility. This means that they are always visible although the legality of their use will depend upon the context. So

```
package P is
   type Roman_Digit is ('I', 'V', 'X', 'L', 'C', 'D', 'M');
end P;

with P;
package Q is
...
   Five: constant P.Roman_Digit := 'V';
...
end Q;
```

is allowed in Ada 95 although in Ada 83 we would have had to write `P.'V'` or alternatively supplied a use clause for `P`.

The visibility rules now also take account of the introduction of child packages. Most of the changes are straightforward but there is one interesting interaction with subunits. Consider

```
package P is
   ...
end P;

package P.Q is
```

```
   ...
end P.Q;

package body P is
   Q: Integer;   -- legal
   procedure Sub is separate;
end P;

with P.Q;          -- illegal
separate(P)
procedure Sub is
   ...
end Sub;
```

The declaration of Q in the body of P is permitted because the body does not have a with clause for P.Q. But the with clause makes the subunit illegal because it could otherwise see both P.Q the child package and P.Q the variable in the body, and they are not overloadable.

8.2 Use Clauses

As mentioned in Part One the use of operators in Ada 83 caused problems. Many groups of users recognized that the use clause could make programs hard to understand because the origin of identifiers became obscure. Accordingly many organizations banned use clauses. This meant that either operators of user defined types had to be called with prefix notation such as Complex."+"(P, Q) or else had to be locally renamed.

This difficulty also occurred with predefined operators. Thus given an access type T declared in a package P, it was very annoying to find that one could not write

```
with P;
procedure Q is
   X: P.T;
begin
   ...
   if X /= null then
   ...
end Q;
```

but had to provide a use clause for P or a renaming for "=" or write the diabolical

```
if P."/="(X, null) then
```

This problem is overcome in Ada 95 by the introduction of the use type clause which just provides visibility of the operators of a type and thereby allows them to be used in the natural infixed form. This ensures that a use package clause is not needed and hence the full name is still required for other identifiers.

The introduction of child units causes some extension to the rules for packages. As explained in Chapter 10, child units are treated like separately compiled but logically nested units. Like nested units, the name of a child mentioned in a with clause becomes directly visible when the logically enclosing parent package is specified in a use clause.

And so, using the example from II.8

```
with OS.File_Manager;
procedure Hello is
   use OS;   -- makes File_Manager directly visible
             -- as well as other declarations in package OS
```

```
      File: File_Descriptor :=
               File_Manager.Open("Hello.Txt", File_Manager.Write_Only);
   begin
      File_Manager.Write(File, "Hello world.");
      File_Manager.Close(File);
   end Hello;
```

8.3 Renaming

To enhance the usefulness of renaming, the body of a subprogram may be provided by a renaming declaration.

If the subprogram declaration is in a package specification while the subprogram definition via a renaming is in a package body, the renaming must be of a subprogram that has subtype conformance (see 6.2) with the subprogram's declaration. This ensures that the caller of the subprogram will perform the correct constraint checks on the actual parameters, and pass the parameters following the correct calling convention seeing only the subprogram's specification.

A normal subprogram renaming requires only mode conformance. This kind of conformance is too weak for a renaming provided in the body. Given only mode conformance, the caller might perform constraint checks that were too stringent or too lax, and might pass parameters following the wrong calling conventions, putting them in the wrong place on the stack, or in the wrong register.

We considered requiring subtype conformance for all subprogram renaming. However, this introduces upward incompatibilities, particularly given the existing equivalence between generic formal subprogram matching and renaming. Furthermore, it is not always possible to statically match the subtype of a formal parameter of a subprogram, if the subprogram is implicitly declared as part of the type definition. In particular, if the subprogram is derived from a parent type, then the formal parameter subtypes have the constraints that were present on the parent type's subprogram. If the derived type definition itself imposes a constraint, then it is likely that the constraint on the formal parameter of the derived subprogram is actually looser than the constraint on the first subtype of the derived type. This means there is no nameable subtype that has constraints as loose as those on the formal parameter.

In the case of a primitive operation of a tagged type, renaming will cause a new slot in the dispatch table to be created if the renaming is itself primitive (that is in the same package specification as the type). If the original primitive operation is overridden then the renamed view will naturally depend upon whether renaming occurs before or after the overriding. Consider

```
   package P is
      type T is tagged ...;
      function Predefined_Equal(X, Y: T) return Boolean renames "=";
      function "="(X, Y: T) return Boolean:  -- overrides
      function User_Defined_Equal(X, Y: T) return Boolean renames "=";
   end P;
```

where we have renamed the predefined equality both before and after overriding it. Both renamings create new slots which are then initialized with the current meaning of equality. That for `Predefined_Equal` thus refers to the predefined equal whereas that for `User_Defined_Equal` refers to the overridden version. The consequence is that renaming can be used to hang on to an old primitive operation irrespective of whether that old operation is subsequently overridden. Such a renaming is itself a distinct primitive operation which could later be overridden for any subsequently derived type.

On the other hand a renaming which is not a primitive operation will not create a new slot but will simply refer to the operation at the point of the renaming. Thus if `User_Defined_Equal` is declared in a distinct package `Q` (after `P`), then it will not be primitive but will still refer to the overridden operation. This will occur even if the overriding is in the private part and thus not visible to `Q`. For a further discussion see [AARM 8.5.4].

This ability of renaming to create a new slot may be considered surprising because the general purpose of renaming is simply to create a new name for an existing entity; but there is of course no new entity being created but just a different way of accessing an existing entity.

Another very useful change is the ability to rename a library unit as a library unit. (It was possible to rename a library unit in Ada 83 but only as a local unit.) Library unit renaming is particularly important with the hierarchical library; this is discussed in detail in 10.1.2.

A related change is the ability to rename a generic unit. Curiously enough this was not permitted in Ada 83 although most other entities could be renamed. Thus we can write

```
generic package Enum_IO renames Ada.Text_IO.Enumeration_IO;
```

as mentioned in [RM95 8.5.5].

In order to prevent difficulties with generic children, a child of a generic parent (such a child must be generic) can only be renamed if the renaming occurs *inside* the declarative region of its parent; it could be renamed as a child. This is consistent with the rules regarding the instantiation of such generic child units mentioned in II.8.

8.4 Other Improvements

The overload resolution rules of Ada 83 were confusing and unclear and this section of the reference manual has been completely rewritten.

An important new rule is the preference rule for operators and ranges of root numeric types (see 3.3). Briefly this says that an ambiguity can be resolved by preferring the operator of the root type. This rule coupled with automatic conversion from a universal type removes the need for the special rules in Ada 83 regarding universal convertible operands.

As an example, consider

```
C: constant := 2 + 3;
```

which was allowed in Ada 83 because the expression was required to be universal and so no question of ambiguity arose. It is not ambiguous in Ada 95 either but for different reasons; the expression `2 + 3` is considered to be of type *root_integer* (it could otherwise be `Integer` or `Long_Integer`). The *root_integer* is then converted implicitly to *universal_integer* as required for the initial value.

The special rule regarding ranges in loops and for array indexes (which are in the distinct syntactic category discrete_subtype_definition) which result in them by treated as of type `Integer` if no specific subtype mark is specified is changed in Ada 95. The new rule is that if the range resolves to be of the type *root_integer* then it is taken to be of type `Integer`.

One outcome of all this is that we can now write

```
for I in -1 .. 100 loop
```

as we have already mentioned in Part One. The interpretation is that `-1` resolves to *root_integer* because of the preference rules and then the special rule just mentioned is used so that the range is finally treated as of type `Integer`.

8.5 Requirements Summary

Many of the changes in this area of the language are aimed at making the language more precise and easier to understand as requested by the general requirement

R2.2-B(1) — Understandability

and we note in particular that the example in [DoD 90 A.2.3] concerning visibility of literals and operations has been addressed and satisfied.

The related requirement

R2.2-C(1) — Minimize Special Case Restrictions

discussed in [DoD 90 A.3.12] contains the example of negative literals in loops which has also been satisfied.

The requirement

R2.2-A(1) — Reduce Deterrents to Efficiency

is addressed by the elimination of the problem of returning task objects explicitly mentioned in the requirement in [DoD 90].

9 Tasking

As explained in Part One, experience with Ada 83 showed that although the innovative rendezvous provided a good overall model for task communication, nevertheless it had not proved adequate for problems of shared data access.

Accordingly, as outlined in Chapter II, this part of the language is considerably enhanced with the major additions being the introduction of protected types, the requeue statement and asynchronous transfer of control. In this chapter we add further examples and discussion to that already given. The main changes to the core language are

- The protected type is introduced as a new form of program unit with distinct specification and body in a similar style to packages and tasks.

- The requeue statement is added to provide preference control and thereby overcome various race conditions which could arise in Ada 83.

- There is a new form of delay statement which allows waiting for an absolute rather than a relative time. This overcomes problems of poor timing which in essence are caused by a race condition with the clock.

- There is a new form of the select statement which gives asynchronous transfer of control. This can be used to program mode changes.

- The description of the rules regarding the abort statement is improved.

- A task (like a protected type) may now have discriminants and a private part.

- A minor improvement is that entries and representation clauses may now occur in any order in a task specification.

In addition to the above changes there are further packages, pragmas and attributes plus requirements on implementations described in the Systems Programming and Real-Time Systems annexes. These relate to matters such as scheduling and priorities, task identification, shared variable access, accuracy of timing, interrupt handling and the immediacy of the abort statement. For further details on these topics the reader is referred to Part Three.

9.1 Protected Types

In Ada 83, the rendezvous was used for both inter-task communication and for synchronizing access to shared data structures. However, the very generality of the rendezvous means that it has a relatively high overhead. Ada 95 overcomes this problem by introducing a low overhead, data-oriented synchronization mechanism based on the concept of protected objects.

From the client perspective, operating on a protected object is similar to operating on a task object. The operations on a protected object allow two or more tasks to synchronize their manipulations of shared data structures.

From the implementation perspective, a protected object is designed to be a very efficient conditional critical region (see 9.1.3). The protected operations are automatically synchronized to allow only one writer or multiple readers. The protected operations are defined using a syntax similar to a normal subprogram body, with the mutual exclusion of the critical region happening automatically on entry, and being released automatically on exit.

We considered many different approaches to satisfying the needs for fast mutual exclusion, interrupt handling, asynchronous communication, and various other common real-time paradigms. We settled on the protected object construct because it seems to provide a very efficient building block, which is flexible enough to implement essentially any higher-level synchronization mechanism of interest.

Some of the features that make protected objects attractive as a building block are:

Scalability. Protected objects enable the implementation of a synchronization mechanism that scales smoothly from a single processor to a multiprocessor. There is no built-in bias to a monoprocessor or to a multiprocessor.

Adaptability. Additional protected operations may be added to a protected type without the need to modify the existing operations. Approaches that use explicit signals and conditions rather than the conditional barrier approach generally need each operation to be aware of every state of interest, and explicitly signal all possible waiting tasks.

Modularity. All of the operations of a given critical region are identified in the specification of the protected type, and their implementations are encapsulated within the body of the protected type. The directly protected data components are encapsulated within the private part of the protected type, and all of the interesting states are identified explicitly via entry barrier conditions within the body of the protected type.

Efficiency. The size and initialization requirements of a protected object are known at compile time of the client, because all entries and data components are declared in the specification. This enables protected objects to be allocated statically or directly on the stack, rather than via dynamic allocation, and to be initialized in-line. No extra context switches are required to service waiting clients, since the task changing the state may directly execute the entry bodies whose barriers become true. Non-queued locking may be used to implement the mutual exclusion of a protected object because no blocking is permitted during the execution of a protected operation.

Expressiveness. The specification of a protected type makes explicit distinctions between read-only operations (functions), read-write operations (procedures), and possibly blocking operations (entries). This distinction is vital in analyzing a real-time program for correctness, including freedom from deadlock.

Compatibility. The entry call remains the primary mechanism for blocking a task until some condition is satisfied. An entry call on another task is blocked until the corresponding entry in the task is open. An entry call on a protected object is blocked until the corresponding entry barrier is true. An entry call on a task is completed when the rendezvous finishes and an entry call on a protected object is completed when the entry body finishes; possibly with intermediate blockings as part of a requeue in both cases.

From the caller's perspective, an entry call involves a possible initial blocking, one or more intermediate blockings, and then a return. During the entry call, data may be transferred via the parameters. By using the entry call interface for both protected objects and tasks, the existing conditional and timed entry call mechanisms are applicable.

Interrupt Handling. A protected procedure is very well suited to act as an interrupt handler for a number of reasons; they both typically have a short bounded execution time, do not arbitrarily block, have a limited context and finally they both have to integrate with the priority model. The nonblocking critical region matches the needs of an interrupt handler, as well as the needs of non-interrupt-level code to synchronize with an interrupt handler. The entry barrier construct allows an interrupt handler to signal a normal task by changing the state of a component of the protected object and thereby making a barrier true.

Of the many other approaches we considered for supporting data-oriented synchronization, none could match this set of desirable features.

Like task types, protected types are limited types. Because protected objects are specifically designed for synchronizing access from concurrent tasks, a formal parameter must always denote the same protected object as the corresponding actual parameter and so pass by reference is required (a copy would not preserve the required atomicity).

A protected type may have discriminants, to minimize the need for an explicit initialization operation, and to control composite components of the protected objects including setting the size of an entry family. A discriminant can also be used to set the priority and identify an interrupt.

The other data components of a protected object must be declared in the specification of the type to ensure that the size is known to the compiler when the type is used by a caller. However these components are only accessible from protected operations defined in the body of the protected type and thus are declared in the private part of the protected type.

The protected operations may be functions, procedures, or entries. All the entries must be declared in the protected type specification to ensure that space needed for entry queues is included when allocating the protected object. Entries which are not required to be visible to external clients can be declared in the private part. Additional functions and procedures may be declared in the private part or body of the protected unit, for modularizing the implementation of the operations declared in the specification.

An example of a counting semaphore implemented as a protected type is given in II.9. This example illustrates the three kinds of protected operations: functions, procedures, and entries. Functions provide read-only access (which may be shared) to the components of the protected object. Procedures provide exclusive read-write access to the components. Entries have an entry barrier that determines when the operation may be performed. The entry body is performed with exclusive read-write access to the components, once the barrier becomes true due to the execution of some other protected operation. It is important to observe that the evaluation of barrier expressions is also performed with exclusive access to the protected object.

The counting semaphore might be used as follows

```
Max_Users: constant := 10;
-- limit number of simultaneous users of service
User_Semaphore: Counting_Semaphore(Max_Users);

procedure Use_Service(P: Param) is
begin
   -- wait if too many simultaneous users
   User_Semaphore.Acquire;
   begin
       Perform_Service(P);
   exception
      when others =>
         -- always release the semaphore for next user.
         User_Semaphore.Release;
         raise;
   end;
   -- release the semaphore for others
   User_Semaphore.Release;
end Use_Service;
```

This example illustrates that a semaphore can be implemented as a protected object. However, in essence this often constitutes abstraction inversion since it leaves the responsibility for releasing the semaphore in the hands of the user (unless we use a controlled type as illustrated in 7.4). It is precisely to avoid such possibilities that protected types are provided as an intrinsic syntactic form inherent in the language. Semaphores (like the goto statement) are very prone to misuse and should be avoided where possible. However, there are occasions when they are useful and this example shows an implementation.

As mentioned, some of the protected operations declared in the specification may be declared after the reserved word **private**. This makes these operations callable only from within the protected unit. Task types may similarly have a private part, so that certain task entries may be hidden from a direct call from outside the task (they can be called by its sub-tasks or via a requeue statement).

(Alternative structures were also considered and rejected. One was that the private part of a protected unit be visible to the enclosing package. However, this was considered confusing, and felt to be inconsistent with the visibility of the private part of a subpackage. We also considered splitting the protected type (and task) specification into two separate parts, with the private operations and data components declared in a second part included inside the private part of the enclosing package. However, this seemed like an unnecessary extra syntactic complexity, so we finally adopted the simpler suggestion from two Revision Requests (RR-0487, RR-0628) of using private within the specification to demarcate the private operations.)

Each entry declared in the specification of a protected type must have an entry body. An entry body includes a barrier condition following the reserved word **when**; the barrier condition must be true before the remainder of the entry body is executed.

The entry body for an entry family must specify a name for the entry family index, using an iterator notation (**for** I **in** discrete_subtype_definition). We considered a simpler syntax (I: discrete_subtype_definition) but opted for the iterator notation to avoid ambiguity with the formal parameter part.

An entry barrier is not allowed to depend on parameters of the entry, but it may depend on the entry family index, or any other data visible to the entry body. This rule ensures that all callers of the same entry see the same barrier condition, allowing the barrier to be checked without examining individual callers. Without this rule, each caller of a given entry would have to be treated separately, since each might have a different effective barrier value. Rather than entry "queues" one would essentially have a single large "bag" of callers, all of which would have to be checked on each protected object state change.

For flexibility, entry barriers may depend on data global to the protected object. This allows part of the data managed by the protected object to be outside it, if this is necessary due to some other program structure requirements. However, the barriers are only rechecked after completing a protected procedure or entry body, so asynchronous changes to global data have no immediate effect on the eligibility of a caller waiting on an entry queue. For efficiency, implementations may assume that the only meaningful changes to data referenced in an entry barrier of some protected object take place within a protected operation of that protected object.

We considered disallowing references to globals in a barrier expression. However, that would also have disallowed the use of functions (which might reference globals) or the dereferencing of access values. Such a rule was felt to be too complex to implement, and too restrictive when dealing with data types implemented with access types, such as a linked list.

The semantics for protected types are described in terms of mutual exclusion (except that protected functions may execute concurrently). In addition, as the final step of a protected action, the entry queues are serviced before allowing new calls from the outside to be executed. In this context, a protected "action" is the whole sequence of actions from locking to unlocking and thus comprises a series of one or more of:

- a call on a protected subprogram from outside the protected unit,
- the execution of an entry body,

- the addition or removal of a call from an entry queue.

Servicing the entry queues is required if any change has been made that might affect the value of a barrier expression. First, the barriers for the non-empty entry queues must be reevaluated. If at least one such barrier evaluates to true, some eligible caller must be selected, and the corresponding entry body must be executed. The barriers are then reevaluated once more, and this process continues until all non-empty entry queues have a false barrier. The barriers may be evaluated, and the entry bodies executed, by any convenient thread of control. It need not be the thread of the original caller. This flexibility allows for the most efficient implementation, minimizing unnecessary context switches. (For details on how the choice of caller is made see the Real-Time Systems annex.)

While executing a protected operation of some protected object, a task cannot call a potentially blocking operation for any reason, though it may release the mutual exclusive access to the protected object by being requeued. Disallowing blocking while executing a protected operation allows a nonqueued locking mechanism to be used to implement the mutual exclusion. If blocking were allowed, then a queued locking mechanism would be required, since potential callers might attempt to get the lock while the current holder of the lock is blocked. Another advantage is that conditional calls are more meaningful.

In the simplest monoprocessor environment, protected object mutual exclusion can be implemented by simply inhibiting all task preemption. If multiple priorities are supported, then rather than inhibiting all preemption, a ceiling priority may be established for the protected object (see the Ceiling Priorities section of the Real-Time Systems annex). Only tasks at this ceiling priority or below may use the protected object, meaning that tasks at priorities higher than the ceiling may be allowed to preempt the task performing the protected operation while still avoiding the need for a queued lock.

In a multiprocessor environment, spin waiting may be used in conjunction with the ceiling priority mechanism to implement a non-queued protected object lock.

By disallowing blocking within a protected operation and by also using the ceiling priority mechanism, unbounded priority inversion can be avoided. The generality that might be gained by allowing blocking would inevitably result in an increase in implementation complexity, run-time overhead, and unbounded priority inversion.

To simplify composability, protected operations may call other non-blocking protected operations (protected procedures and functions). A direct call on a protected subprogram within the same protected type does not start a new protected action, but is rather considered to be part of the current action. It is considered an error if, through a chain of calls going outside the protected object, a call is made back to the same protected object. The effect is implementation-defined, but will generally result in a deadlock. We considered disallowing all subprogram calls from a protected operation to a subprogram defined outside the protected type, but this seemed unnecessarily constraining, and to severely limit composability.

9.1.1 Summary of Mechanism

Protected types provide a low-level, lightweight, data-oriented synchronization mechanism whose key features are

- A protected object has hidden components; these components are intended to be shared among multiple tasks. The protected operations of the protected object provide synchronized access to the components.

- Protected types have three kinds of protected operations: protected functions, protected procedures, and entries. Protected functions and protected procedures are known as protected subprograms.

- Protected procedures provide exclusive read-write access to the components. Protected functions, since they cannot change the components of the protected object, may be optimized to use a shared read-only lock.

- Protected entries also provide exclusive read-write access to the components, but in addition, they specify a barrier, which is a Boolean expression that generally depends on the components of the protected object. The Run-Time System ensures that the barrier is true before allowing a protected entry call to proceed.

- Each protected object has a conceptual lock associated with it. (This lock may sometimes be an actual one, or can instead be implemented using the ceiling priorities model, see the Real-Time Systems annex). At the start of a protected operation, the calling task seizes the lock. Evaluation of barriers, execution of protected operation bodies, and manipulation of entry queues (see below) are all done while the lock is held. On a multiprocessor, the intended implementation of locks uses busy waiting. (Other, more specialized algorithms, are allowed).

- There is a queue associated with each protected entry. Tasks wait in the queue until the entry's barrier becomes `True`. If the barrier is already `True` when the entry call first seizes the lock, then it is executed immediately; the queue is not used. While waiting in the queue, a task does not hold the lock.

- A requeue statement is allowed in an entry body (and an accept statement). The effect of this statement is to return the current caller back to the queue, or to place it on another, compatible, entry queue.

- Ceiling priorities may be associated with protected objects as described in the Real-Time Systems annex. The ceiling rules prevent the priority-inversion phenomenon, and ensure freedom from deadlocks on single-processor systems. For further details see Part Three of this rationale.

9.1.2 Examples of Use

Protected types combine high efficiency with generality and can be used as building blocks to support various common real-time paradigms. In this section we discuss three examples

- An implementation of indivisible counters showing how protected subprograms are used.

- An implementation of persistent signals showing how protected entries are used.

- A generic bounded buffer showing how protected types can be used within a generic package.

A non-generic form of the bounded buffer and an implementation of transient (broadcast) signals will be found in II.9. Examples of the use of the requeue mechanism are shown in 9.2.

We observe that protected types allow Ada 95 to support these and other real-time paradigms with a smaller overall change to the language than the alternative approach where each problem is solved with its own distinct feature.

In the following examples, we refer to the lock as an actual object with lock and release operations. This, of course, is not required, and is simply used for ease of presentation.

The first very simple example shows a counter that is shared among multiple tasks

```
protected Counter is
   procedure Increment(New_Value: out Positive);
private
   Data: Integer := 0;
end Counter;

protected body Counter is
   procedure Increment(New_Value: out Positive) is
   begin
      Data := Data + 1;
      New_Value := Data;
   end Increment;
end Counter;
```

The counter is initialized to zero. A task may increment it by calling the `Increment` procedure

```
Counter.Increment(New_Value => X);
```

If N tasks do this, each exactly once, they will each get a unique value in the range 1 to N. Note that without the synchronization provided by the protected type, multiple simultaneous executions of `Increment` might cause unpredictable results. With the protected type, a task that calls `Increment` will first seize the lock, thus preventing such simultaneous executions.

Since there are no entries in this example, there are no queues. The protected type consists, in essence, of a lock and the component `Data`.

If we want to define many `Counter` objects, we would change the above example to declare a protected type instead of a single protected object as follows

```
protected type Counter_Type is
   ... -- same as before
end Counter_Type;

Counter_1, Counter_2: Counter_Type; -- declare two counters

type Many is array (1 .. 1000) of Counter_Type;

X: Many;                               -- declare many counters
```

It is important to note that a lock is associated with each protected object and not with the type as a whole. Thus, each of the objects in the above example has its own lock, and the data in each is protected independently of the others.

This simple example has a short, bounded-time algorithm; all it does is increment the value and assign it to the **out** parameter. This is typical of the intended use of protected types. Because the locks might be implemented as busy-waits (at least on multiprocessors), it is unwise to write an algorithm that might hold a lock for a long or unknown amount of time. A common approach where extensive processing is required would be to just record the new state, under the protection of the lock, and do the actual processing outside the protected body.

The next example shows a persistent signal. In this example, tasks may wait for an event to occur. When the event occurs, some task whose job it is to notice the event will "signal" that the event has occurred. The signal causes exactly one waiting task to proceed. The signal is persistent in the sense that if there are no tasks waiting when the signal occurs, the signal persists until a task invokes a wait, which then consumes the signal and proceeds immediately. Multiple signals when no tasks are waiting are equivalent to just one signal.

It is interesting to note that persistent signals are isomorphic to binary semaphores; the wait operation corresponds to P, and the signal operation corresponds to V.

```
protected Event is
   entry Wait;          -- wait for the event to occur
   procedure Signal;    -- signal that the event has occurred.
private
   Occurred: Boolean := False;
end Event;

protected body Event is

   entry Wait when Occurred is
   begin
      Occurred := False; -- consume the signal
   end Wait;

   procedure Signal is
   begin
      Occurred := True;  -- set Wait barrier True
   end Signal;

end Event;
```

A task waits for the event by calling

```
Event.Wait;
```

and the signalling task notifies the happening of the event by

```
Event.Signal;
```

whereupon the waiting task will proceed.

There are two possibilities to be considered according to whether the call of `Wait` or the call of `Signal` occurs first.

If a call of `Wait` occurs first, the task will seize the lock, check the barrier, and find it to be `False`. Therefore, the task will add itself to the entry queue, and release the lock. A subsequent `Signal` will seize the lock and set the `Occurred` flag to `True`. Before releasing the lock, the signalling task will check the `Wait` entry queue. There is a task in it, and the barrier is now `True`, so the body of `Wait` will now be executed, setting the flag to `False`, and the `wait`ing task released. Before releasing the lock, the process of checking entry queues and barriers is repeated. This time, the `Wait` barrier is `False`, so nothing happens; the lock is released, and the signalling task goes on its way.

If, on the other hand, a call of `Signal` occurs first, then the task will seize the lock, set the flag to `True`, find nothing in the entry queues, and release the lock. A subsequent `Wait` will seize the lock, find the barrier to be `True` already, and proceed immediately with its body. The barrier is now `False`, so the waiting task will simply release the lock and proceed.

Important things to note are

- Protected subprograms do not have barriers. Protected entries always have barriers. `Signal` is a protected procedure, because it never needs to block while waiting for the state of the protected type to change; that is, it needs no barrier. `Wait` is an entry, because it needs to block until the flag is set before proceeding.

- The protected entry queues are not used for tasks waiting for access to the protected data. They are used for tasks waiting for an entry barrier to become true; that is, they are waiting for the state of the protected type to change in order to satisfy some user-specified condition. The lock associated with the protected object synchronizes access to the shared data. The same lock also protects the entry queues themselves; the queues may be considered as part of the shared data.

- Barriers are associated with entry queues, not with individual tasks calling the entries. Therefore, the number of barriers to be evaluated can never be more than the number of entries of the protected object. (Note that the size of an entry family can depend on a discriminant, so the number of entries can vary from object to object of the type.)

- Entry barriers are reevaluated only at well-defined points: in particular, when a call is first made and when a protected procedure or entry associated with that protected object has just finished, but before it has released the lock.

- Protected functions do not change the state of the protected type, and so should not change barrier values. Therefore, entry barriers are not reevaluated when a protected function has just finished.

- The lock stays locked during the actions that happen at the end of a protected procedure or entry (that is, checking the queues for non-empty status, evaluating barriers, removing a task from a queue, and executing the entry body). This means that tasks already on the queues get preference over tasks that are trying to seize the lock.

- Barriers, like protected operation bodies, should contain only short, bounded-time expressions. In typical examples, the barrier simply tests a `Boolean` flag, or checks the number of elements in an entry queue. The programmer has complete control over worst-case waiting times; this worst-case analysis can be done by inspecting the algorithms in the protected bodies, the barriers, and the maximum number of tasks that can be expected to be waiting on each barrier.

- The language does not specify which task executes a particular barrier or entry body. One can think of barriers and entry bodies as being executed by the Run-Time System, not by any particular task. Certain restrictions on what may appear in a barrier or an entry body imply that it does not matter which task does the work. (For example, `Current_Task` cannot be called, see Systems Programming annex.) This is done to ensure that no context switches are required; when one task finishes executing an operation, the task can immediately execute any others that have become ready, without having to switch to the context of the "correct" task.

Our next example is a generic form of a bounded buffer. In this example a protected object provides conditional critical regions, which allow the abstraction to be used safely by multiple tasks.

```
generic
   type Item is private;
   Mbox_Size: in Natural;
package Mailbox_Pkg is

   type Item_Count is range 0 .. Mbox_Size;
   type Item_Index is range 1 .. Mbox_Size;
   type Item_Array is array (Item_Index) of Item;

   protected type Mailbox is
      -- put a data element into the buffer
      entry Send(Elem: Item);
      -- retrieve a data element from the buffer
```

```
      entry Receive(Elem: out Item);
   private
      Count    : Item_Count := 0;
      Out_Index: Item_Index := 1;
      In_Index : Item_Index := 1;
      Data     : Item_Array;
   end Mailbox;

end Mailbox_Pkg;
```

This example illustrates a generic mailbox abstraction. The protected type has two entries, which insert and retrieve items to and from the mailbox buffer. Like a private type, the data components of the protected type are of no concern outside the body. They are declared in the specification so that a compiler can statically allocate all the space required for an instance of the protected type.

The body of the mailbox package is as follows

```
package body Mailbox_Pkg is
   protected body Mailbox is
      entry Send(Elem: Item) when Count < Mbox_Size is
           -- block until there is room in the mailbox
      begin
         Data(In_Index) := Elem;
         In_Index := In_Index mod Mbox_Size + 1;
         Count := Count + 1;
      end Send;

      entry Receive(Elem: out Item) when Count > 0 is
           -- block until there is something in the mailbox
      begin
         Elem := Data(Out_Index);
         Out_Index := Out_Index mod Mbox_Size + 1;
         Count := Count - 1;
      end Receive;
   end Mailbox;
end Mailbox_Pkg;
```

As we saw in the non-generic example in II.9, `Send` waits until there is room for a new `Item` in the mailbox buffer. `Receive` waits until there is at least one `Item` in the buffer. The semantics of protected records guarantee that multiple tasks cannot modify the contents of the mailbox simultaneously.

A minor point is that the type `Item_Array` has to be declared outside the protected type. This is because type declarations are not allowed inside a protected type which generally follows the same rules as records. Allowing types within types would have introduced additional complexity with little benefit. For elegance we have also declared the types `Item_Count` and `Item_Index`.

9.1.3 Efficiency of Protected Types

Protected types provide an extremely efficient mechanism; the ability to use the thread of control of one task to execute a protected operation on behalf of another task reduces the overhead of context switching compared with other paradigms. Protected types are thus not only much more efficient than the use of an agent task and associated rendezvous, they are also more efficient than traditional monitors or semaphores in many circumstances.

As an example consider the following very simple protected object which implements a single buffer between a producer and a consumer task.

```
protected Buffer is
   entry Put(X: in Item);
   entry Get(X: out Item);
private
   Data: Item;
   Full: Boolean := False;
end;

protected body Buffer is
   entry Put(X: in Item) when not Full is
   begin
      Data := X;  Full := True;
   end Put;

   entry Get(X: out Item) when Full is
   begin
      X := Data;  Full := False;
   end Get;
end Buffer;
```

This object can contain just a single buffered value of the type `Item` in the variable `Data`; the boolean `Full` indicates whether or not the buffer contains a value. The barriers ensure that reading and writing of the variable is interleaved so that each value is only used once. The buffer is initially empty so that the first call that will be processed will be of `Put`.

A producer and consumer task might be

```
task body Producer is
begin
   loop
      ... -- generate a value
      Buffer.Put(New_Item);
   end loop;
end Producer;

task body Consumer is
begin
   loop
      Buffer.Get(An_Item);
      ... -- use a value
   end loop;
end Consumer;
```

In order to focus the discussion we will assume that both tasks have the same priority and that a run until blocked scheduling algorithm is used on a single processor. We will also start by giving the processor to the task `Consumer`.

The task `Consumer` will issue a call of `Get`, acquire the lock and then find that the barrier is false thereby causing it to be queued and to release the lock. The `Consumer` is thus blocked and so a context switch occurs and control passes to the task `Producer`. This sequence of actions can be symbolically described by

```
Get(An_Item);
   lock
      queue
   unlock
switch context
```

The task `Producer` issues a first call of `Put`, acquires the lock, successfully executes the body of `Put` thereby filling the buffer and setting `Full` to `False`. Before releasing the lock, it reevaluates the barriers and checks the queues to see whether a suspended operation can now be performed. It finds that it can and executes the body of the entry `Get` thereby emptying the buffer and causing the task `Consumer` to be marked as no longer blocked and thus eligible for processing. Note that the thread of control of the producer has effectively performed the call of `Get` on behalf of the consumer task; the overhead for doing this is essentially that of a subprogram call and a full context switch is not required. This completes the sequence of protected actions and the lock is released.

However, the task `Producer` still has the processor and so it cycles around its loop and issues a second call of `Put`. It acquires the lock again, executes the body of `Put` thereby filling the buffer again. Before releasing the lock it checks the barriers but of course no task is queued and so nothing else can be done; it therefore releases the lock.

The task `Producer` still has the processor and so it cycles around its loop and issues yet a third call of `Put`. It acquires the lock but this time it finds the barrier is false since the buffer is already full. It is therefore queued and releases the lock. The producer task is now blocked and so a context switch occurs and control at last passes to the consumer task. The full sequence of actions performed by the producer while it had the processor are

```
Put(New_Item);
   lock
      Data := New_Item;  Full := True;
      scan: and then on behalf of Consumer
      An_Item := Data;  Full := False;
      set Consumer ready
   unlock
Put(New_Item);
   lock
      Data := New_Item:  Full := True;
      scan: nothing to do
   unlock
Put(New_Item);
   lock
      queue
   unlock
switch context
```

The consumer task now performs a similar cycle of actions before control passes back to the producer and the whole pattern then repeats. The net result is that three calls of `Put` or `Get` are performed between each full context switch and that each call of `Put` or `Get` involves just one lock operation.

This should be contrasted with the sequence required by the corresponding program using primitive operations such as binary semaphores (mutexes). This could be represented by

```
package Buffer is
   procedure Put(X: in Item);
   procedure Get(X: out Item);
private
   Data: Item;
   Full: Semaphore := busy;
   Empty: Semaphore := free;
end;

package body Buffer is
   procedure Put(X: in Item) is
   begin
      P(Empty);
```

```
         Data := X;
         V(Full);
      end Put;

      procedure Get(X: out Item) is
      begin
         P(Full);
         X := Data;
         V(Empty);
      end Get;
   end Buffer;
```

In this case there are two lock operations for each call of `Put` and `Get`, one for each associated semaphore action. The behavior is now as follows (assuming once more that the consumer has the processor initially). The first call of `Get` by the consumer results in the consumer being suspended by `P(Full)` and a context switch to the producer occurs.

The first call of `Put` by the producer is successful, the buffer is filled and the operation `V(Full)` clears the semaphore upon which the consumer is waiting. The second call of `Put` is however blocked by `P(Empty)` and so a context switch to the consumer occurs. The consumer is now free to proceed and empties the buffer and performs `V(Empty)` to clear the semaphore upon which the producer is waiting. The next call of `Get` by the consumer is blocked by `P(Full)` and so a context switch back to the producer occurs.

The net result is that a context switch occurs for each call of `Put` or `Get`. This contrasts markedly with the behavior of the protected object where a context switch occurs for every three calls of `Put` or `Get`.

In conclusion we see that the protected object is much more efficient than a semaphore approach. In this example it as a factor of three better regarding context switches and a factor of two better regarding locks.

Observe that the saving in context switching overhead depends to some degree on the run-until-blocked scheduling and on the producer and consumer being of the same priority. However, the saving on lock and unlock overheads is largely independent of scheduling issues.

The interested reader should also consult [Hilzer 92] which considers the more general bounded buffer and shows that monitors are even worse than semaphores with regard to potential context switches.

9.1.4 Relationship with Previous Work

Protected types are related to two other synchronization primitives: the conditional critical region and the monitor. The protected type has been incorporated in a way that is compatible with Ada's existing task types, entries, procedures and functions.

In 1973, Hoare proposed a synchronization primitive called a conditional critical region [Hoare 73] with the following syntax

```
   region V when barrier do
      statements
   end;
```

where the barrier is a Boolean expression, and V is a variable. The semantics of the construction may be described as follows [Brinch-Hansen 73]:

> *When the sender enters this conditional critical region, the [barrier expression] is evaluated. If the expression is true the sender completes the execution of the critical region ... But if the expression is false, the sender leaves the critical*

region temporarily and enters an anonymous queue associated with the shared variable V.

However, Brinch-Hansen pointed out a disadvantage with conditional critical regions [Brinch-Hansen 73]:

Although [conditional critical regions] are simple and well-structured, they do not seem to be adequate for the design of large multiprogramming systems (such as operating systems). The main problem is that the use of critical regions scattered throughout a program makes it difficult to keep track of how a shared variable is used by concurrent processes. It has therefore been recently suggested that one should combine a shared variable and the possible operations on it in a single, syntactic construct called a monitor.

This thus led to the monitor which has a collection of data and subprogram declarations. In Ada terms, the subprograms declared in the visible part of a monitor, and which are therefore visible outside the monitor, are guaranteed to have exclusive access to the data internal to the monitor. The monitor may also have some variables known as condition variables. These condition variables are like semaphores, in that they have `Wait` and `Signal` operations. A `Wait` operation waits for a matching `Signal` operation. Hoare introduces monitors with somewhat different syntax, but with equivalent semantics [Hoare 74].

The problem with monitors as discussed in [IBFW 86] is that the Signal and Wait operations suffer from the usual difficulties of using semaphores; they can easily be misused and the underlying conditions are not easy to prove correct.

The Ada 83 rendezvous followed CSP [Hoare 78] by providing a dynamic approach to the problem and one which clarified the guarding conditions. However, as we have discussed, the rendezvous has a heavy implementation overhead through the introduction of an intermediary task and can also suffer from race conditions.

Ada 95 protected objects are an amalgam of the best features of conditional critical regions and monitors: they collect all the data and operations together, like monitors, and yet they have barriers, like conditional critical regions. The barriers describe the required state that must exist before an operation can be performed in a clear manner which aids program proof and understanding. Protected objects are very similar to the shared objects of the Orca language developed by Bal, Kaashoek and Tanenbaum [Bal 92].

9.2 The Requeue Statement

Components such as complex servers or user-defined schedulers often need to determine the order and the timing of the service provided by entry or accept bodies based on the values of various controlling items. These items may be local to the server and dependent on its own state, be an attribute of the client or the controlled task, or be global to the system. In addition, these items may often change from the time the entry call is made to the time the selection decision is itself finally made.

For fairly simple cases — that is when the items are known to the caller, do not change from the time of call, and have a relatively small discrete range — the entry family facility of Ada 83 might suffice (see [Wellings 84], [Burger 87], [Burns 87]). However, when those restrictions do not hold, a more powerful mechanism is often needed.

Entry queue selection is sometimes called preference control. Many papers discussing preference control have appeared in the literature [Elrad 88, Wellings 84, Burns 89]. Preference control arises in applications like resource allocation servers, which typically grant satisfiable requests and queue up unsatisfiable requests for later servicing. Wellings, Keefe and Tomlinson [Wellings 84] were unable to find a good way to implement such servers in Ada 83.

An intrinsic provision within a language of the full expressive power to describe the various forms of preference controls would require an elaborate semantic structure and a complex (and potentially large) run-time support system.

Instead, we have chosen to provide a single and simple statement in Ada 95 which allows the programmer to construct the desired control algorithms based on the balance of needs of specific applications. This is the requeue statement and as we saw in Part One, it enables an entry call to be requeued on another or even the same entry.

In order for the server to gain access to the caller's parameters there is no need to resume the caller and to require it to initiate another entry call based on the results of the first; it may simply be moved to another entry queue. An alternate approach that was considered required the caller to first query the state of the server, and then to initiate an entry call with appropriate parameters (presumably using a specific family member index) to reflect the server's state. This approach suffers from the potential of race conditions, since no atomicity is guaranteed between the two calls (another caller may be serviced and the state of the server may be changed), so the validity of the second request which is based on the first, may be lost.

In the case of protected entry calls, exclusive access is maintained throughout the period of examining the parameters and doing the requeue; in the case of accept bodies, the server task controls its own state and since it can refuse to accept any intermediate calls, the atomicity is also maintained.

The requeue statement may be specified as **`with abort`**. In Ada 83, after a rendezvous had started, there was no way for the caller to cancel the request (or for a time-out to take effect — a time-out request is present only until the acceptor *starts* the service). There was, of course, good reason for this behavior; after the service has commenced, the server is in a temporary state, and removing the caller asynchronously can invalidate its internal data structures. In addition, because of by-reference parameters, the acceptor must maintain its ability to access the caller's data areas (such as the stack). If the caller "disappears", this might result in dangling references and consequent disaster.

However, in some cases, deferring the cancellation of a call is unacceptable, in particular when the time-out value is needed to control the amount of time until the service is completed (as opposed to just started). With the addition of asynchronous transfer of control to Ada 95, the same situation can arise if the caller is "interrupted" and must change its flow of control as soon as possible.

Since there is not a single best approach for all applications, and since no easy work-around exists, the **`with abort`** is provided to allow the programmer to choose the appropriate mechanism for the application concerned. In general, when cancellation during a requeue is to be allowed, the server will "checkpoint" its data-structures before issuing requeue **`with abort`**, in such a way that if the caller is removed from the second queue, the server can continue to operate normally. When this is not possible, or when the cancellation during a requeue is not required, a simple requeue will suffice, and will hold the caller until the service is fully completed.

The requeue statement is designed to handle two main situations

- After an accept statement or entry body begins execution, it may be determined that the request cannot be satisfied immediately. Instead, there is a need to requeue the caller until the request can be handled.

- Alternatively, part of the request may be handled immediately, but there may be additional steps in the process that need to be performed at a later point.

In both cases, the accept statement or entry body needs to relinquish control so that other callers may be handled or other processing may be performed; the requeue enables the original request to be processed in two or more parts.

The requeue statement allows a caller to be "requeued" on the queue of the same or some other entry. The caller need not be aware of the requeue and indeed the number of steps required to handle a given operation need not be visible outside the task or protected type. The net effect is that the server can be more flexible, while presenting a simple single interface to the client.

As part of the requeue, the parameters are neither respecified nor reevaluated. Instead, the parameters are carried over to the new call directly. If a new parameter list were specifiable, then it might include references to data local to the accept statement or entry body itself. This would cause problems because the accept statement or entry body is completed as a consequence of the requeue and its local variables are thus deallocated. Subtype conformance is thus required between the new target entry (if it has any parameters) and the current entry. This allows the same representation to be used for the new set of parameters whether they are by-copy or by-reference and also eliminates the need to allocate new space to hold the parameters. Note that the only possibility other than passing on exactly the same parameters is that the requeued call requires no parameters at all.

As a first example of requeue, the reader is invited to look once more at the example of the broadcast signal which we first met in II.9 and which we now repeat for convenience.

As in the previous signal example in 9.1.2, tasks wait for an event to occur. However, this is a broadcast signal because when the event is signaled, all waiting tasks are released, not just one. After releasing them, the event reverts to its original state, so tasks can wait again, until another signal. Note also, that unlike the previous example, the event here is not persistent. If no tasks are waiting when the signal arrives, it is lost.

```
protected Event is
   entry Wait;
   entry Signal;
private
   entry Reset;
   Occurred: Boolean := False;
end Event;

protected body Event is

   entry Wait when Occurred is
   begin
      null;         -- note null body
   end Wait;

   entry Signal when True is  -- barrier is always true
   begin
      if Wait'Count > 0 then
         Occurred := True;
         requeue Reset;
      end if;
   end Signal;

   entry Reset when Wait'count = 0 is
   begin
      Occurred := False;
   end Reset;

end Event;
```

The intended use is that tasks wait for the event by calling

```
Event.Wait;
```

and another task notifies them that the event has occurred by calling

```
Event.Signal;
```

and this causes all currently waiting tasks to continue, and the event to be reset, so that future calls to `Event.Wait` will wait.

The example works as follows. If a task calls `Event.Wait`, it will first seize the protected object lock. It will check `Occurred`, find it to be `False`, add itself to the entry queue, and release the lock. Several tasks might add themselves to the queue in this manner.

Later, the signalling task might call `Event.Signal`. After seizing the lock, the task will execute the entry body (since its barrier is `True`). If no tasks are currently waiting, the task exits without updating the flag. Otherwise, it sets the flag to indicate that the event has occurred, and requeues itself on the `Reset` entry. (`Reset` is declared in the private part, because it is not intended to be used directly by clients of the protected object.)

Before releasing the lock, the signalling task will check the queues. The barrier for `Wait` is now `True`. A task is chosen from the `Wait` queue, and allowed to proceed. Since the entry body for `Wait` does nothing, the flag will not change. (See the Real-Time Systems annex for the detailed rules for choosing among the tasks waiting in entry queues.) This sequence of events will be repeated until the entry queue for `Wait` is empty. When the `Wait` queue is finally empty (that is `Wait'Count` equals 0), the barrier of `Reset` is `True`, and the `Reset` body is executed, thereby resetting the flag. The queues are now empty, so the protected object lock is released. Note that implementations can optimize null entry bodies by releasing waiting tasks in one operation, when the barrier is true.

Because the steps described in the last two paragraphs are executed with the protected object locked, any other tasks that try to `Wait` or `Signal` on that object during that time will have to wait for the queues to be emptied, as explained above. Furthermore, there are no race conditions because the value of the barrier cannot be changed between the time it is evaluated and the time the corresponding entry body is executed.

The check for now-true barriers happens whenever the state of the protected object might have changed and, of course, before releasing the lock; that is, they happen just after executing the body of a protected procedure or protected entry.

In summary

- A requeue statement is only allowed in an entry body and an accept statement. The target entry may be (possibly the same entry) in the same task or protected object or in a different task or protected object. All combinations are possible.

- Any actual parameters to the original call are passed to the new entry; therefore, the new entry must have the same parameter profile, or else no parameters at all.

In the broadcast example, the requeue statement prevents a race condition that might otherwise occur. For example, if the signalling task were required to call `Signal` and `Reset` in sequence, then the releasing of waiting tasks would no longer be atomic. A task that tried to `Wait` in between the two calls of the signalling task, might have been released as well. It might even be the same task that was already released once by that `Signal`.

We noted in Part One that this example was for illustration only and could be programmed without using requeue. Here is a possible solution which uses the `Count` attribute in barriers for both `Wait` and `Signal`.

```
protected Event is
   entry Wait;
   entry Signal;
end Event;

protected body Event is
   entry Wait when Signal'Count > 0 is
   begin
      null;
   end Wait;
```

```
      entry Signal when Wait'Count = 0 is
      begin
         null;
      end Signal;
   end Event;
```

When the `Wait` entry is called, the caller will be blocked until some caller is enqueued on the `Signal` entry queue. When the `Signal` entry is called, the caller is queued if there are any waiters, then all the tasks on the `Wait` entry queue are resumed. The signaler is then dequeued and the entry call is complete. If there are no waiters when `Signal` is called, it returns immediately.

This is an interesting solution. It works because the event of joining a queue is a protected action and results in the evaluation of barriers (just as they are evaluated when a protected procedure or entry body finishes). Note also that there is no protected data (and hence no private part) and that both entry bodies are null); in essence the protected data is the `Count` attributes and these therefore behave properly. In contrast, the `Count` attributes of task entries are not reliable because the actions of adding and joining task entry queues are not performed in any protected manner.

9.2.1 Preference Control

Many real-time applications require preference control, where the ability to satisfy a request depends on the parameters passed in by the calling task and often also on the internal state of the server. Examples are

- The server must serve higher-priority requests first, where the priority of the request is passed as an entry parameter.

- A particular entry call is used to request any of several resources, where some resources might be available, while others are in use. An entry parameter indicates which of the resources the task is requesting.

- Several copies of a resource may be allocated at once, where the calling task passes in the number of required resources. A specific instance of this situation is a memory allocator, where the resource is a block of storage units. The calling task asks for a particular number of storage units, and must wait until a contiguous portion of memory of at least the right size is available. It might be that a request for 100 storage units can be satisfied, whereas a request for 1000 storage units cannot.

- The server is controlling a device that might be ready to serve some requests but not others.

We now consider an example of the last situation. We have a disk device with a head that may be moved to different tracks. When a calling task wants to write to the disk at a particular place, the call may proceed immediately if the disk head is already on the right track. Otherwise, the disk manager tells the disk device to move the head. When the disk has moved the head, it generates an interrupt. While waiting for the interrupt, the calling task is blocked.

Preference control can be implemented in Ada 95 using the requeue statement. The entry call proceeds whether it can be immediately satisfied or not. Then the server checks to see whether the request can be immediately satisfied by looking at the parameters. If it can, the request is processed, and the entry returns. If not, the request is requeued to another entry, to wait until conditions change.

The preference control in our example is simple. We can satisfy requests for the current disk track, and queue the others. Since the disk address is passed in as an entry parameter, some calls to the `Write` entry can proceed, while others cannot.

```
protected Disk_Manager is
   entry Write(Where: Disk_Address; Data: Disk_Buffer);
      -- write data to the disk at the specified address
   entry Read(Where: Disk_Address; Data: out Disk_Buffer);
      -- read data from the disk at the specified address
   procedure Disk_Interrupt;
      -- called when the disk has interrupted, indicating
      -- that the disk head has moved to the correct track
private
   entry Pending_Write(Where: Disk_Address; Data: Disk_Buffer);
   entry Pending_Read(Where: Disk_Address; Data: out Disk_Buffer);

   Current_Disk_Track: Disk_Track_Address := ...;
      -- track where the disk head currently is.
   Operation_Pending: Boolean := False;
      -- is an incomplete Read or Write operation pending?
   Disk_Interrupted: Boolean := False;
      -- has the disk responded to the move command with
      -- an interrupt?
end Disk_Manager;
```

In order to write on the disk, a task calls `Disk_Manager.Write`, passing the disk address and data as parameters. The `Read` operation is similar but the full details are omitted. The body of the protected object is as follows

```
protected body Disk_Manager is

   procedure Disk_Interrupt is
   begin
      Disk_Interrupted := True;  -- release pending operations
   end Disk_Interrupt;

   entry Pending_Write(Where: Disk_Address; Data: Disk_Buffer)
      when Disk_Interrupted is
   begin
      Current_Disk_Track := Where.Track;
          -- we know that the disk head is at the right track.
      ... -- write Data to the disk
      Operation_Pending := False;
          -- allow Reads and Writes to proceed
   end Pending_Write;

   entry Write(Where: Disk_Address; Data: Disk_Buffer)
      when not Operation_Pending is
   begin
      if Where.Track = Current_Disk_Track then
         ... -- write Data to the disk
      else
         ... -- tell the disk to move to the right track
         Disk_Interrupted := False;
         Operation_Pending := True;
             -- prevent further Reads and Writes
         requeue Pending_Write; -- wait for the interrupt
      end if;
   end Write;
```

```
      entry Pending_Read(Where: Disk_Address; Data: out Disk_Buffer)
         when Disk_Interrupted is
      begin
         ... -- similar to Pending_Write
      end Pending_Read;

      entry Read(Where: Disk_Address; Data: out Disk_Buffer)
         when not Operation_Pending is
      begin
         ... -- similar to Write
      end Read;

   end Disk_Manager;
```

The `Write` operation checks whether the disk head is already on the right track. If so, it writes the data and returns. If not, it sends a command to the disk telling it to move the head to the right track, and then requeues the caller on `Pending_Write`. It sets a flag to prevent intervening `Write` and `Read` operations. When the disk has completed the move-head command, it interrupts, causing the `Disk_Interrupt` operation to be invoked. The `Disk_Interrupt` operation sets the flag that allows the `Pending_Write` operation to proceed.

We do not specify here how `Disk_Interrupt` gets called when the interrupt occurs. It might be attached directly to the interrupt, or some other interrupt handler might call it or set a flag in some other protected object that causes `Disk_Interrupt` to be called.

A real disk manager would be more complicated; it would probably allow multiple pending requests, sorted by track number, the actual reading and writing might be interrupt driven (in addition to the disk head movement), and so on. But this simple version nevertheless illustrates the key features of preference control.

The following points should be noted

- One might think that an obvious way to implement preference control would be to make the entry barrier depend on the parameters of the entry. However, that is not allowed in order to permit efficient implementations and to avoid complex semantics for the user. Because the barriers do not depend on the formal parameters, the value of the barrier is the same for all callers of the same entry. This means that there is only one barrier value per entry, not one per entry call, so evaluation of barriers can be efficient. If the barrier expression can be evaluated in a bounded amount of time, as is usually the case, the programmer can calculate total worst-case barrier evaluation times based on the worst-case barrier evaluation times of each of the entries.

- It is important that the decisions of when to service a request happen inside a protected operation, because these decisions are based on the protected object's local data, which, of course must be protected. The Ada 95 requeue mechanism achieves such protection.

- The example works properly in the presence of abort and asynchronous transfer of control. For example, if a writing task is aborted while it is waiting on the `Pending_Write` queue, the abort will be deferred until after `Pending_Write` has been executed. On the other hand, the programmer might wish to allow the task to be aborted earlier. In that case, the requeue statement would take the form

  ```
  requeue Pending_Write with abort;
  ```

and then the protected body would have to be written in such a way that callers can silently disappear from the `Pending_Write` queue without disruption. (The barriers would have to depend upon `Pending_Write'Count`.)

- In Ada 83, an extra "agent" task was required to "hold" the call in examples such as this. In order to correctly handle abort and asynchronous transfer of control, and still handle multiple outstanding requests, the agent tasks had to be created dynamically, as necessary. Such agent tasks were too expensive for many applications, and their timing behavior was rather unpredictable.

- Other methods in Ada 83 required exporting several entries that had to be called in a particular order. This violated information-hiding principles, and caused race conditions, because events could occur between the multiple calls.

  The information-hiding objection could be answered by putting the task in a package, and putting the correct pattern of entry calls in a procedure exported from the package, thus enforcing the required protocol. But then, the resulting exported procedure could not be used in timed, conditional, and selective entry calls.

  Other solutions to the race problems generally required the requesting task to poll the server, which was inefficient and nondeterministic, in addition to being error-prone.

9.3 Timing

Ada 83 was unhelpful in the area of timing by notionally providing only one clock that could be used directly for delaying a task or timing an operation. Furthermore, the Ada 83 delay statement required a duration, rather than a wakeup time, making it difficult for a task to wake up at perfectly regular intervals as explained in II.10.

In Ada 95, the delay statement is augmented by a delay until statement. The delay until statement takes the wakeup time rather than a duration as its argument. Furthermore, the Real-Time Systems annex defines the package `Ada.Real_Time` which contains an additional time type `Real_Time.Time` with an accompanying function `Real_Time.Clock`, which may be used in a delay until statement.

The type `Real_Time.Time` is intended to represent a real-time clock with potentially finer granularity than the time-of-day clock associated with `Calendar.Time`. Furthermore, the value returned by `Real_Time.Clock` is guaranteed to be monotonically non-decreasing, whereas the time-of-day `Calendar.Clock` may jump forward or backward due to resetting of the time by a human operator (perhaps in response to a change of time-zone or daylight saving).

The following example shows a task that awakens each night at midnight and performs some logging function.

```
task body At_Midnight is
   One_Day : constant Calendar.Day_Duration := 86_400.0;
   Now     : Calendar.Time := Calendar.Clock;
   Midnight: Calendar.Time := Calendar.Time_Of(
      Year    => Calendar.Year(Now),
      Month   => Calendar.Month(Now),
      Day     => Calendar.Day(Now),
      Seconds => 0.0);
         -- truncate current time to most recent midnight
begin
   loop
      Midnight := Midnight + One_Day;
      delay until Midnight;
```

```
         Log_Data;
      end loop;
   end At_Midnight;
```

Since the delay until expression specifies an absolute time rather than a time interval, there is no opportunity for preemption during the calculation of the interval, and therefore the delay will expire at precisely the time that is specified.

Note furthermore that since the delay is written in terms of the time-of-day clock in the package `Calendar`, if the time-of-day clock is changed to daylight saving time (or perhaps the cruise liner moves over a time zone), the delay expiration time might be according to the new setting of the clock (although this is not guaranteed).

As a further example we present a task that polls a device every 10 milliseconds.

```
   task body Poll_Device is
      use Ada;
      Poll_Time: Real_Time.Time := time to start polling;
      Period: constant Real_Time.Time_Span :=
                                Real_Time.Milliseconds(10);
   begin
      loop
         delay until Poll_Time;
         ... -- Poll the device
         Poll_Time := Poll_Time + Period;
      end loop;
   end Poll_Device;
```

In this case the `Poll_Device` task polls the device every 10 milliseconds starting at the initial value of `Poll_Time`. The period will not drift, as explained above for the `At_Midnight` example. We use `Real_Time.Time` instead of `Calendar.Time` in this example, because we do not wish to be sensitive to possible changes to the time-of-day clock.

The existing (relative) delay statement only takes a value of the type `Duration`; the basis for relative delays is not necessarily that of the clock in the package `Calendar` and should be monotonic. The general idea is that relative delays should not be disturbed by a shift in the time base. A ten minute delay still means just that even if the clock moves forward.

Finally note that in Ada 95 the package `Calendar` is a child of `Ada`. For compatibility it is also renamed as `Standard.Calendar` (all such existing library units in Ada 83 are similarly renamed for compatibility).

9.4 Asynchronous Transfer of Control

Asynchronous transfer of control was identified as an important requirement for Ada 95 (Requirement R5.3-A(1)). In Ada 83, the only way to asynchronously change the execution path of a task was to abort it. However, in many applications, it is desirable that an external event be able to cause a task to begin execution at a new point, without the task as a whole being aborted and restarted.

As an example of asynchronous transfer of control, consider an interactive program where the user may choose to terminate a given operation and begin a new one. This is normally signaled by typing a special key or hitting a special button associated with the controlling input device. The user does not want the entire context of the running program to be lost. Furthermore, for a long-running system, it is important that the resources associated with the interrupted processing be reclaimed. This implies that some mechanism for "cleaning up" be available as part of the asynchronous transfer process. Finally, if the abortable operation is updating some global data structure, it is essential to temporarily defer any asynchronous transfers until after the update is complete.

As was briefly explained in II.10, Ada 95 has a form of select statement with an abortable part and a triggering alternative to support asynchronous transfer of control. We showed a simple example where a computation was abandoned if it could not be completed within a stated period of time.

In essence the triggering statement and the abortable part execute in parallel and whichever finishes first causes the other to be abandoned.

If the triggering statement completes before the abortable part, then the abortable part is abandoned and control passes to the sequence of statements following the triggering statement. On the other hand, if the abortable part completes before the triggering statement then the triggering alternative is abandoned.

The important point is that we only need one thread of control. Waiting for a delay or waiting on an entry queue do not require a separate thread. Moreover, when a task entry is accepted it is the called task which executes the accept statement and even in the case of a protected entry it will often be another task which executes the entry body. So the abortable part can generally continue during the execution of the triggering statement. It is only when the entry call finally returns (or the delay expires) that the abortable part has to be abandoned. For full details of the mechanism see [RM95 9.7.4].

By supporting asynchronous transfer of control as a form of select statement, several useful properties are provided

- The statements that are abortable are clearly bracketed in the abortable part.

- The asynchronous transfer of control is directly tied to the acceptance of an entry call or the expiration of a delay. This allows the transfer to occur without requiring an additional task to explicitly signal the occurrence of the triggering event; this is in contrast to what is possible with abort.

- Nesting of abortable regions (which are potentially "sensitive" to different events) and protecting code in these regions from interruption, is naturally achieved by the select construct and protected types.

- The asynchronous transfer cannot be mistakenly redirected by a local handler, as might happen with a mechanism based on asynchronous exceptions.

Here is an example of a database transaction using asynchronous transfer of control. The database operation may be cancelled by typing a special key on the input device. However, once the transaction has begun (is committed), the operation may not be cancelled.

```
with Ada.Finalization; use Ada;
package Txn_Pkg is
   type Txn_Status is (Incomplete, Failed, Succeeded);
   type Transaction is new Finalization.Limited_Controlled with
      private;
   -- Transaction is a controlled type as discussed in 7.4
   procedure Finalize(Txn: in out Transaction);
   procedure Set_Status(Txn: in out Transaction;
                        Status: Txn_Status);
private
   type Transaction is new Finalization.Limited_Controlled with
      record
         Status: Txn_Status := Incomplete;
         pragma Atomic (Status);
         ... -- More components
      end record;
end Txn_Pkg;
```

```
package body Txn_Pkg is

   procedure Finalize(Txn: in out Transaction) is
   begin
      -- Finalization runs with abort and ATC deferred
      if Txn.Status = Succeeded then
         Commit(Txn);
      else
         Rollback(Txn);
      end if;
   end Finalize;

   procedure Set_Status(Txn: in out Transaction);
                        Status: Txn_Status) is
   begin
       Txn.Status := Status;
   end Set_Status;

end Txn_Pkg;
```

The package might be used as in the following

```
declare
    Database_Txn: Transaction;
     -- declare a transaction, will commit or abort
     -- during finalization
begin
   select
      -- wait for a cancel key from the input device
      Input_Device.Wait_For_Cancel;
       -- the Status remains Incomplete, so that
       -- the transaction will not commit
   then abort
       -- do the transaction
      Read(Database_Txn, ...);
      Write(Database_Txn, ...);
      ...
      Set_Status(Database_Txn, Succeeded);
      -- set status to ensure the transaction is committed
   exception
      when others =>
         Put_Line("Operation failed with unhandled exception");
         -- set status to cause transaction to be aborted
         Set_Status(Database_Txn, Failed);
   end select;
   -- Finalize on Database_Txn will be called here and,
   -- based on the recorded status, will either commit or
   -- abort the transaction.
end;
```

This illustrates the use of controlled types and asynchronous transfer of control. At the end of the block, the `Finalize` operation is called and this will uniquely either rollback the transaction or commit to it. Note in particular the use of the pragma `Atomic`; this is described in the Systems Programming annex. Note also that the `Finalization` is always performed with abort and ATC deferred so that no unfortunate interactions can occur.

The final example shows how asynchronous transfer of control can be used in a real-time application. `Current_Coordinates` is periodically updated with a new set of computed coordinates. A user task (not shown) can call `Read` as needed to get the most recently computed coordinates, which might then be used to control an external device.

```
protected Current_Coordinates is
   procedure Update(New_Val: Coordinates);
     -- used by the computing task only
   function Read return Coordinates;
     -- used by whoever needs the result
private
   Current_Value: Coordinates;
end Current_Coordinates;

protected Controller is
   entry Wait_For_Overrun;
     -- called by the computing task
   procedure Set_Overrun;
     -- called by an executive or interrupt handler
private
   Overrun_Occurred: Boolean := False;
end Controller;
```

The protected object `Current_Coordinates` provides mutually exclusive access to the most recently calculated coordinates. The protected object `Controller` provides an entry for detecting overruns, designed to be called in the triggering alternative of an asynchronous select statement as shown below.

The following is the body of the `Calculate` task, which loops, progressively improving the estimate of the coordinates, until its time allotment expires, or the estimate stabilizes.

```
task body Calculate is
   Problem: Problem_Defn;
begin
   Get_Problem(Problem);
   select
      Controller.Wait_For_Overrun;   -- triggering alternative
   then abort
      declare
         Answer: Coordinates := Initial_Value;
         Temp  : Coordinates;
      begin
         Current_Coordinates.Update(Answer);
         loop  -- loop until estimate stabilizes
            Temp := Answer;
            Track_Computation.Improve_Estimate(Problem, Answer);
            Current_Coordinates.Update(Answer);
            exit when Distance(Answer, Temp) <= Epsilon;
         end loop;
      end;
   end select;
end Calculate;
```

The `Calculate` task sets the value of the `Current_Coordinates` initially, and then repeatedly calls `Track_Computation.Improve_Estimate`, which is presumably a time-consuming procedure that calculates a better estimate of the coordinates. `Calculate` stops looping when it decides that the estimate has stabilized. However, it may be that `Improve_Estimate` takes too long, or the system undergoes a mode change that requires the use of the current best estimate. There is presumably an executive or interrupt handler that notices such a situation and calls `Controller.Set_Overrun`. When that happens, the `Calculate` task does an asynchronous transfer of control thereby ending the computation loop.

We now show a possible (partial) implementation of the Improve_Estimate subprogram. It depends on some work area that has a dynamic size, that can be allocated, lengthened, and deallocated. Improve_Estimate allocates the work area, and tries to compute the result. However, the computation of the result may fail, requiring a larger work area. Therefore, Improve_Estimate loops until it succeeds or the time expires or some resource is exhausted.

```
with Ada.Finalization; use Ada;
package body Track_Computation is
   -- This package includes a procedure Improve_Estimate for
   -- progressively calculating a better estimate of the coordinates.

   -- buffer is used for a work area to compute new coordinates
   type Buffer_Size is range 0 .. Max;
   type Buffer is ...
   type Buffer_Ptr is access Buffer;

   type Work_Area is new Finalization.Limited_Controlled with
      record
         Buf: Buffer_Ptr;
      end record;

   -- these procedures allocate a work area of a given size,
   -- and reallocate a longer work area
   procedure Allocate_Work_Area(
        Area: in out Work_Area;
        Size: in Buffer_Size) is ...
   procedure Lengthen_Work_Area(
        Area  : in out Work_Area;
        Amount: in Buffer_Size) is ...

   -- this procedure is called automatically on scope exit,
   -- and deallocates the buffer designated by Area.Buf
   procedure Finalize(Area: in out Work_Area) is ...

   procedure Improve_Estimate(
        Problem: in Problem_Defn;
        Answer : in out Coordinates) is
      -- calculate a better estimate, given the old estimate
      Initial_Size: Buffer_Size := Estimate_Size(Problem);
      -- compute expected work area size, based on problem definition
      Work_Buffer: Work_Area;
   begin
      Allocate_Work_Area(Work_Buffer, Initial_Size);
      loop
         begin
            ... -- compute better estimate
            Answer := ...;
            exit;  -- computation succeeded
         exception
            when Work_Area_Too_Small =>
               -- the Problem requires a larger work area
               Lengthen_Work_Area(Work_Buffer, Size_Increment);
               -- now loop around to try again
         end;
      end loop;
      -- Work_Buffer is automatically deallocated by
      -- finalization on exit from the scope
   end Improve_Estimate;
```

```
    end Track_Computation;
```

Since it is important that the work area be deallocated when the asynchronous transfer of control occurs, `Work_Area` is derived from `Finalization.Limited_Controlled` so that a `Finalize` procedure can be defined. This provides automatic clean up on scope exit.

Note that the `Calculate` task does not (and should not) need to know about the implementation details of `Improve_Estimate`. Therefore, it is not feasible to put the call on `Finalize(Work_Buffer)` in `Calculate`. Furthermore, `Allocate_Work_Area` might not use a normal Ada allocator. It might be allocating from some static data structure. In any case, it is important to reclaim the resources allocated to the work area when the processing is complete or aborted.

Aborts and asynchronous transfers of control are deferred when a task is performing a protected subprogram or entry call, or during an initialization or finalization operation on an object of a controlled type. The programmer has complete control over the amount of code that should be placed in such abort-deferred regions. Typically, such regions should be kept short.

9.5 The Abort Statement

In Ada 95, it is essential that use of the abort statement, and, more importantly, the asynchronous select statement with its abortable part, not result in corruption of global data structures. In Ada 83, abort was deferred for a calling task while it was engaged in a rendezvous. This allowed the rendezvous to complete normally so that the data structures managed by the accepting task were not left in an indeterminate state just because one of its callers was aborted.

For Ada 95, we have generalized this deferral of abort to include the time during which calls on protected operations are being serviced, and the initialization, assignment and finalization of controlled types (see 7.4). (However, we recall that requeue with abort allows a server to override deferral if desired as explained in 9.2.)

Without deferral of abort, any update of a global data structure becomes extremely unsafe, if not impossible. Ultimately all updates are forced into a two-phase approach, where updates are first performed into unused storage, and then the final commitment of a change involves a single atomic store, typically of a pointer of some sort. Such an approach can be extremely cumbersome, and very inefficient for large data structures. In most cases, it is much simpler and efficient to selectively defer asynchronous transfers or aborts, rather than to allow them at any moment.

In addition to deferring abort, it is important to be able to reclaim resources allocated to constructs being aborted. The support for user-defined initialization and finalization of controlled types provides the primitives necessary to perform appropriate resource reclamation, even in the presence of abort and asynchronous transfers.

Reclaiming local storage resources is of course important. However, releasing resources is even more critical for a program involved in communicating with an external subsystem, such as a remote database or other server. For a short-lived program, running on a conventional time-shared operating system, with no contact with external subsystems, it might be argued that there is no need to provide user-defined finalization that runs even when the task is aborted or is "directed" to execute an asynchronous transfer of control. However, for a long-running program, with limited memory, and which is possibly communicating with external subsystems, it is crucial that relatively local events like an asynchronous transfer not undermine global resource management.

In general, the discussion in [RM95] is unified so that aborting a task and aborting a sequence of statements (as in ATC) are described together.

9.6 Tasking and Discriminants

In Ada 95, we have generalized discriminants so that they are applicable to task types and protected types as well as to records. This allows tasks and protected objects to be parameterized when they are declared.

An example of a protected type with a discriminant is the `Counting_Semaphore` in II.9. The discriminant indicates the number of items in the resource being guarded by the semaphore.

Discriminants of tasks can be used to set the priority, storage size and size of entry families of individual tasks of a type. In the case of storage size this is done with the new pragma `Storage_Size` by analogy with the pragma `Priority`. (Note that an attribute definition clause could only be applied to all tasks of a type; the use of such an attribute definition clause for setting the storage size is now obsolescent.)

Of more importance is the ability to indicate the data associated with a task; this obviates the need for an initial rendezvous with a task and can eliminate or at least reduce any bottleneck in the parallel activation of tasks.

For example, in a numerical application we might have an array of tasks each of which works on some data. In each case the data will be shared with an adjacent task and so can be conveniently accessed through a protected object. The tasks do not therefore need to communicate with each other directly but just with the protected objects. We might write

```
subtype Data_Range is Integer range 0 .. 1000;
subtype Task_Range is
      Data_Range range Data_Range'First+1 .. Data_Range'Last-1;

protected type Data_Point is
   procedure Put(New_Value: in Data);
   procedure Get(Current_Value: out Data);
private
   -- the protected data
end;
...
The_Data: array (Data_Range) of Data_Point;

function Next_One return Task_Range;
...
task type Computer(Index: Task_Range := Next_One);

The_Tasks: array (Task_Range) of Computer;
```

where we assume that the data at the end-points of the range is fixed (the boundary conditions) and so no task is associated with these.

Successive calls of the function `Next_One` deliver the unique values throughout the task range. This guarantees that each task has a unique value of the discriminant `Index` although this might not correspond to its index position in the array since the components can be initialized in any order. However, they are not permitted to be initialized in parallel and so there is no need for the function `Next_One` to take any action to prevent parallel calls. Each task can then use its discriminant `Index` to access the data in the protected objects.

An access discriminant is particularly useful for indicating the data associated with a task. We could write

```
type Task_Data is
   record
      ...   -- data for task to work on
   end record;

task type Worker(D: access Task_Data) is
```

```
   ...
end;
```

and then inside the body of `Worker` we can get hold of the data via the access discriminant `D`. The data is associated with a particular task in its declaration

```
Data_For_Joe: aliased Task_Data := ...

Joe: Worker(Data_For_Joe'Access);
```

where we note that the data has to be marked as **aliased**. An advantage of access discriminants is that they are constants and cannot be detached from the task; hence the task and its data are securely bound together. We recall from 3.7.1 that there are consequently no access checks on the use of an access discriminant.

An alternative approach is to embed the task inside the data. We can then use the self-referential technique described in 4.6.3.

```
type Task_And_Data is limited
   record
      ... -- some data
      Jobber: Worker(Task_And_Data'Access);
   end record;
```

We can use similar techniques with protected objects. The data logically protected by a protected object need not be directly inside the protected object. It could be indicated by an access discriminant. For example

```
type Resource is
   record
      Counter: Integer;
      ...
   end record;

protected type Guardian(R: access Resource) is
   procedure Increment;
   ...
end Guardian;

protected body Guardian is
   procedure Increment is
   begin
      R.Counter := R.Counter + 1;
   end Increment;
   ...
end Guardian;
```

and then within the bodies of protected procedures such as `Increment` we can access the data of type `Resource` in a safe manner. We declare a particular protected object thus

```
My_Resource: aliased Resource := ...
..
My_Object: Guardian(My_Resource'Access);
...
My_Object.Increment;
```

Clearly this approach can be used with any standard protected object such as the mailbox discussed in 9.1.2.

9.6.1 Interaction with OOP

Tasks and protected objects may appear at first sight to be concerned with aspects of programming quite alien to the concepts associated with object oriented programming such as type extension and dispatching. For example, it is not possible to extend a protected type with additional protected operations (this was considered at conflict with other considerations such as efficiency). However, although indeed an orthogonal part of the language, tasks and protected objects work together with tagged types in a remarkable way. Thus we can create structures with both synchronization and extension properties where protected objects or tasks provide the synchronization aspects and tagged types provide extension aspects.

For example, a task or protected object may be a component of a tagged object or conversely may contain a tagged object internally. A powerful construction is where a task or protected object has a class-wide access discriminant that references a tagged object. In this section we give some examples of the use of access discriminants in this way.

The first example illustrates how a task type can provide a template for a variety of related activities with the details filled in with dispatching calls. (Remember from the discussion in 3.7.1 and 4.4.4 that type extension provides a flexible means of parameterizing a general activity such as the use of an iterator.)

Suppose that we wish to perform a number of activities that have the general pattern of performing some job a certain number of times at intervals with perhaps some initial and final actions as well. We can also make provision for a general purpose exception handling mechanism. A suitable task type might be

```
task type T(Job: access Job_Descriptor'Class);

task body T is
begin
   Start(Job);
   for I in 1 .. Iterations(Job) loop
      delay Interval(Job);
      Do_It(Job, I);
   end loop;
   Finish(Job);
exception
   when Event: others =>
      Handle_Failure(Job, Event);
end T;
```

Note carefully that the access discriminant `Job` is class-wide so that dispatching can occur. The various subprograms `Start`, `Iterations`, `Interval`, `Do_It`, `Finish` and `Handle_Failure` are all dispatching operations of the type `Job_Descriptor` and might be declared as follows.

```
package Base_Job is
   type Job_Descriptor is abstract tagged null record;
   procedure Start(J: access Job_Descriptor);
   function Iterations(J: access Job_Descriptor) return Integer
                                                         is abstract;
   function Interval(J: access Job_Descriptor) return Duration
                                                         is abstract;
   procedure Do_It(J: access Job_Descriptor; I: Integer) is abstract;
   procedure Finish(J: access Job_Descriptor);
   procedure Handle_Failure(J: access Job_Descriptor;
                            E: Exception_Occurrence);
end Base_Job;
```

We have made most of the operations abstract so that the user is forced to provide nonabstract versions but have chosen to make `Start` and `Finish` just null since that is an obvious default. A convenient default for `Handle_Failure` would also seem appropriate.

Observe that the various operations dispatch on an access parameter. It would have been possible for the parameters to be just **in** parameters but then the actual parameters would have had to be `Job.all` in the various calls. (If we wanted write access to `Job` then the parameters would have to be **in out** or **access** parameters; see 6.1.2 for a discussion on the merits of one versus the other. Moreover, we might make the parameters of `Iterations` and `Interval` of mode **in** just to emphasize that they are read only operations.)

A demonstration task to output a message ten times with one minute intervals might be produced by writing

```
with Base_Job; use Base_Job;
package Demo_Stuff is
   type Demo is new Job_Descriptor with null record;
   function Iterations(D: access Demo) return Integer;
   function Interval(D: access Demo) return Duration;
   procedure Do_It(D: access Demo; I: Integer);
end;

package body Demo_Stuff is
   function Iterations(D: access Demo) return Integer is
   begin
      return 10;
   end Iterations;

   function Interval(D: access Demo) return Duration is
   begin
      return 60.0;
   end Interval;

   procedure Do_It(D: access Demo; I: Integer) is
   begin
      New_Line; Put("This is number "); Put(I);
   end Do_It;
end Demo_Stuff;
...
The_Demo: Demo;                       -- data for the demo
The_Demo_Task: T(The_Demo'Access);    -- create the task
```

This somewhat pathetic demonstration task always does the same thing since there is actually no data in the type `Demo`. All the object `The_Demo` does is indicate through its tag the particular subprograms to be called by dispatching. Thus the type `Demo` is simply a tag.

A more exciting demonstration might be created by giving the type some components indicating the number of iterations and the interval. The procedure `Start` might then check that the demonstration would not take too long (five minutes would be quite enough!) and, if necessary, by raising an exception cause the demonstration to be cancelled and a suitable message output. This might be written as

```
package Better_Demo_Stuff is
   type Better_Demo is new Job_Descriptor with
      record
         The_Iterations: Integer;
         The_Interval: Duration;
      end record;
   Silly_Demo: exception;
   ...
```

```
end;

package body Better_Demo_Stuff is
   function Iterations(D: access Better_Demo) return Integer is
   begin
      return D.The_Iterations;
   end Iterations;
   ...
   procedure Start(D: access Better_Demo) is
   begin
      if D.The_Iterations * D.The_Interval > 300.0 then
         Raise_Exception(Silly_Demo'Identity, "Sorry; too long");
      end if;
   end Start;

   procedure Handle_Failure(D: access Better_Demo;
                            E: Exception_Occurrence) is
   begin
      Put_Line("Demonstration not executed because: ");
      Put_Line(Exception_Message(E));
   end Handle_Failure;
end Better_Demo_Stuff;
```

For illustrative purposes we have passed the message to be output using the exception message mechanism discussed in 11.2.

Although our example has been merely a simple demonstration nevertheless the approach could be used to much effect in simulations and related applications. By using type extension, unnecessary repetition of common code is avoided.

The next example illustrates how a class of types plus associated protocols originally developed for a non-tasking environment can be encapsulated as a protected type. Again the key is the use of a class-wide access discriminant.

Suppose we have a queuing protocol defined by

```
type Queue is abstract tagged null record;
function Is_Empty(Q: in Queue) return Boolean is abstract;
function Is_Full(Q: in Queue) return Boolean is abstract;
procedure Add_To_Queue(Q: access Queue;
                       X: Queue_Data) is abstract;
procedure Remove_From_Queue(Q: access Queue;
                            X: out Queue_Data) is abstract;
```

Observe that this describes a whole class of queue types. An existent `Queue` need only supply bodies for these four operations. Incidentally `Is_Empty` and `Is_Full` take **in** parameters since they do not modify the queue whereas `Add_To_Queue` and `Remove_From_Queue` take **access** parameters because they do modify the queue. This protocol is similar to the generic package in 4.4.1 except that we have assumed that we know the anticipated specific type when removing items from the queue.

A general protection template is provided by

```
protected type PQ(Q: access Queue'Class) is
   entry Put(X: in Queue_Data);
   entry Get(X: out Queue_Data);
end;

protected body PQ is
   entry Put(X: in Queue_Data) when not Is_Full(Q.all) is
```

```
      begin
         Add_To_Queue(Q, X);
      end Put;

      entry Get(X: out Queue_Data) when not Is_Empty(Q.all) is
      begin
         Remove_From_Queue(Q, X);
      end Get;
   end PQ;
```

Interference between operations on a queue is prevented by the natural mechanism of the protected type. Moreover, the functions `Is_Empty` and `Is_Full` (originally provided to enable the user of the non-tasking protocol to guard against misuse of the queue) can now be sensibly used as barriers to ensure that a user is automatically prevented from misuse of a queue. Note that the particular queue is identified by the access discriminant.

The user can now define a particular implementation of a queue by type extension such as

```
   type My_Queue is new Queue with private;
   function Is_Empty(Q: My_Queue) return Boolean;
   ...
```

and then declare and use a protected queue as follows

```
   Raw_Queue: aliased My_Queue;
   My_Protected_Queue: PQ(Raw_Queue'Access);
   ...
   My_Protected_Queue.Put(An_Item);
```

The unprotected queue object provides the value of the access discriminant for the protected object. Operations upon the protected queue are performed as expected by normal entry calls.

9.7 Other Improvements

An improvement to the syntax allows entries and representation clauses to be in an arbitrary order in a task specification; previously all entries had to precede all representation clauses.

There are a number of other aspects of the control of tasks that are dealt with in the Systems Programming and the Real-Time Systems annexes. These cover topics such as the identification of tasks, task attributes, the handling of interrupts and the control of priorities. For a discussion on these topics the reader is referred to Part Three of this rationale.

9.8 Requirements Summary

The requirement

R2.2-A(1) — Reduce Deterrents to Efficiency

is very clearly addressed by the introduction of protected objects.

The requirements

R5.1-A(1) — Elapsed Time Measurement

R5.1-B(1) — Precise Periodic Execution are met by the introduction of `Real_Time.Time` and the delay until statement (see 9.3).

The requirement

R5.2-A(1) — Alternative Scheduling Algorithms

is generally addressed by the Real-Time Systems annex and is thus discussed in Part Three. A related aspect is selection from entry queues.

The rather general requirement

R5.2-A(2) — Common Real-Time Paradigms

is met variously by the protected type, the requeue statement, and the asynchronous select in the core language and the facilities for priority control in the Real-Time Systems annex.

The requirements

R5.3-A(1) — Asynchronous Transfer of Control

R5.1-C(1) — Detection of Missed Deadlines

are met by the asynchronous select statement discussed in 9.4.

The requirement for

R5.4-A(1) — Non-Blocking Communication

can be met by the use of a protected record as a mailbox buffer.

The study topic

S5.4-B(1) — Asynchronous Multicast

can be met in various ways using protected objects as building blocks.

The requirements

R6.3-A(1) — Interrupt Servicing

R6.3-A(2) — Interrupt Binding

are of great concern to hard real-time programs. They are addressed in detail by the Systems Programming annex.

The study topic

S7.2-A(1) — Managing Large Numbers of Tasks

is addressed by the introduction of task discriminants; see 9.6.

Finally, it should be noted that the two study topics

S7.3-A(1) — Statement Level Parallelism

S7.4-A(1) — Configuration of Parallel Programs

which relate to vector processing are not directly addressed by any features in the language. However, the rules for the interaction between exceptions and optimization have been relaxed [RM95 11.6] so that implementers should be able to add appropriate pragmas to allow statement level parallelism. Mechanisms for controlling the configuration of parallel programs are clearly outside the scope of the language itself.

10 Program Structure and Compilation Issues

There are a number of important changes in this overall structural area of the language. These include the introduction of the hierarchical library which was discussed in some detail in Part One. In this chapter we provide some more examples and also discuss other topics of a structural nature. The main changes are

- The introduction of the hierarchical library provides decomposition of a library unit with possibly distinct views of the hierarchy for the client and implementor.
- The overall program structure is enhanced by the concept of partitions.
- More control of elaboration order is provided.
- A library package is only permitted to have a body if one is required by language rules.
- The mechanism of the program library is no longer specified so precisely. The main issue is that partitions must be consistent.
- Minor changes include the relaxation of a restriction on subunit names.

In addition to the core language changes which introduce the concept of partitions, the Distributed Systems annex describes the concepts of active and passive partitions plus various packages which provide communication between partitions. These are discussed in detail in Part Three.

10.1 The Hierarchical Library

This topic has already been discussed in Sections II.7 and II.8 of Part One where we saw how the introduction of a hierarchical library with public and private child units overcame a number of problems. We recapitulate some of that discussion here in order to bring further insight into the ways in which the hierarchical library may be used.

In Ada 83, there were a number of situations where relatively small changes result in large numbers of recompilations. For example, it was not possible to define an additional operation for a private type without recompiling the package where the type was declared. This forced all clients of the package to become obsolete and thus also need recompiling, even if they did not use the new operation. Massive recompilations could result from what was fundamentally a very small change.

As we have seen, this is overcome in Ada 95 by allowing a library unit package to be effectively extended with child library units. A child library unit is an independent library unit in that it is not visible unless referenced in a with clause. However, a child library unit may be used to define new operations on types defined in the parent package, because the private part and body of the child unit have visibility onto the private declarations of the parent package.

The name of a child library unit indicates its position in the hierarchy. Its name is an expanded name, with the prefix identifying its parent package. Furthermore, when a child library unit is referenced in a with clause, it "looks like" a unit nested in its parent package. This allows the existing naming and visibility rules of nested units to be carried over directly when using child library units.

If a child library unit is not mentioned in a with clause, it is as if it did not exist at all. Adding a new child library unit never causes any immediate recompilation of existing compilation units. Of course, eventually some number of other library units will come to depend on this child library unit, and then recompiling the child will affect these client units. But by distributing the set of operations across multiple children, the number of clients affected by any single change can be kept to a minimum. Furthermore, the with clause provides explicit and detailed indications of interdependences, helping to document the overall structure of the system.

Separate compilation of program unit specifications and bodies is a powerful facility in Ada. It supports good software engineering practice by separating the abstract interface of the unit from its implementation. Clients of the program unit need only know about its specification — changes to the body do not affect such clients, and do not necessitate a client's recompilation. This separation of interface from implementation also applies to private types. Private types have an interface in the visible part of the package in which they are declared and an implementation in the private part of that package. Declarations in the private part were only visible within that private part and in the package body in Ada 83.

For very complex type definitions, the relationship between private types and packages in Ada 83 introduced an unnecessary coupling between abstractions. Consider, for example, a system that implements two private types, `File_Descriptor` and `Executable_File_Descriptor`. The first supports general file operations, and the second supports special file operations for executable files. Thus `Executable_File_Descriptor` might have a write operation that uses scattering writes to write out an entire executable file very quickly.

In Ada 83, if `Executable_File_Descriptor` must have access to the implementation of `File_Descriptor`, perhaps for reasons of performance, then both `File_Descriptor` and `Executable_File_Descriptor` must be defined in the same package. Clients of either `File_Descriptor` or `Executable_File_Descriptor` will depend on this package. If the definition of `Executable_File_Descriptor` is changed, all units that depend on the common package must be recompiled, even those that only utilize `File_Descriptor`. The unnecessary coupling between `Executable_File_Descriptor` and `File_Descriptor` forces more recompilations than are logically necessary.

The following example shows how to alleviate this situation using child library units

```
package File_IO is
   type File_Descriptor is private;
   -- Operations on File_Descriptor...
private
   ...
end File_IO;

package File_IO.Executable_IO is
   type Executable_File_Descriptor is private;
   -- Operations on Executable_File_Descriptor...
private
   ...
end File_IO.Executable_IO;
```

As a child of package `File_IO`, `File_IO.Executable_IO` can use the full declaration of the private type `File_Descriptor` in the declaration of the private type `Executable_File_Descriptor`. Clients of the package `File_IO` do not require recompilation if a child package changes, and new child units can be added without disturbing existing clients.

Another way of looking at the example above is to observe a distinction between the different clients of a package. The traditional clients of a package use the package's visible definitions as their interface. There are other clients that extend the package's abstractions. These extending clients may add functionality to the original abstraction, or export a different interface, or do both. Extending clients will require details of the original package's implementation. Packages that are extending clients are tightly coupled to the original package in terms of implementation, but their logical coupling may be looser — the extending client may be an alternative to the original for other clients. It should be possible to use either package independently. In Ada 95, one or more child library units can share access to the declarations of their parent's private part — to extend their parent's visible interface or provide an alternate view of it.

The distinction between packages that extend another package and packages that simply use the definitions of another package gives rise to the notion of subsystems. A set of packages that share a set of private types can be viewed as a subsystem or subassembly. This concept is recognized by many design methodologies [Booch 87] and is supported by several implementations in their program library facilities. Subsystems are a useful tool for structured code reuse — very large software systems can be designed and built by decomposing the total system into more manageable-sized pieces that are related, but independent, components.

In summary, child library units provide a combination of compilation independence and hierarchical structuring. This combination results in an extremely flexible building block for constructing subsystems of library units that collectively implement complex abstractions and directly model the structure of the subsystems within the language. As an example, the package `OS` in II.8 illustrates the use of a hierarchy of library units in implementing a subsystem representing an operating system interface.

10.1.1 Summary of Mechanism

In Ada 95, a library package (or a generic library package) may have child library units. Child library units have the following properties

- Child units are logically dependent on their parent and have visibility of their parent's visible and private parts.
- Child library units are named like nested units; with an expanded name consisting of a unique identifier and their parent's name as prefix.
- A child may be any kind of library unit including a generic unit or instantiation.
- This structure may be iterated — child units that are packages (or generic packages) may themselves have children, yielding a tree-like hierarchical structure, beginning at a root library unit. Note however that a generic unit may only have generic children.

Child library units may be separately compiled. They may also be separately "withed" by clients. A context clause that names a child unit necessarily names all of the child unit's ancestors as well (with clauses are hence implicit for the ancestors). Within the private part of `File_IO.Executable_IO`, declarations in the private part of `File_IO` are visible. Hence the definition of `Executable_File_Descriptor` can use the full declaration of `File_Descriptor`.

There are two kinds of child units — private child units and public child units. Private children are those declared using the reserved word **private**.

```
private package File_IO.System_File_IO is
   type System_File_Descriptor is private;
private
   ...
end File_IO.System_File_IO;
```

The visibility rules for private child units are designed such that they cannot be used outside the subsystem defined by their parent. A unit's specification may depend on a private child unit only if the unit is itself private and it is a descendant of the original private child's parent. A unit body may depend on a private child unit if it is a descendant of the private child unit's parent.

Public child packages are intended to provide facilities that are available outside the subsystems defined by their parent packages. In the example, `File_IO.Executable_IO` is a public child generally available to clients. Private child packages are intended "for internal use" in a larger definition. In the example, `File_IO.System_File_IO` is meant to be private to the hierarchy of units rooted at `File_IO`.

Child library units observe the following visibility rules.

- A parent unit's visible definitions are visible everywhere in any child unit, whether the unit is public or private.
- A parent unit's private definitions are visible in the private part of a child unit, whether the child unit is public or private.
- A parent unit's private definitions are visible everywhere in a private child unit, since the child package is never visible outside of the parent.
- The entities in a parent's body are never visible in a child unit.

A principal design consideration for child library units was that it should not be possible for a private view to be violated by indirect means such as renaming. As can be seen this has been achieved since a child unit cannot export a parent unit's private definition by renaming it in the visible part.

Note that for root units, private has the effect of making them invisible to the specifications of public root units. Thus private root units concern the internal implementation of the total system and not its public interface. Moreover, we anticipate that some implementations may provide means for one program library to be referenced from another program library, and in this case the private marking might naturally be used to limit visibility across program libraries. See 10.1.5.

In conclusion, the ability to mark a library unit private is extremely valuable in establishing the same separation between interface and implementation at the subsystem level as is provided in Ada 83 at the individual program unit level. The private library units of a hierarchy, plus the bodies of both the private and public library units of the hierarchy, make up the "implementation" of a subsystem. The "interface" to the subsystem is the declaration of the root package in the hierarchy, plus all of the public descendant child library units.

10.1.2 Context Clauses and Renaming

As in Ada 83, a with clause is used to make other library units visible within a compilation unit. In Ada 95, in order to support the identification of a child library unit, the with clause syntax is augmented to allow the use of an expanded name (the familiar dotted notation) rather than just a single identifier.

Once the child library unit has been identified with its full name in the with clause, normal renaming or use clauses may be used within the compilation unit to shorten the name needed to refer to the child.

In addition to renaming from within a unit, renaming may be used at the library unit level to provide a shorter name for a child library unit, or to hide the hierarchical position of a child unit when appropriate. For example, a library unit may be defined as a child unit to gain special visibility on a parent unit, but this visibility may be irrelevant to most users of the child unit.

Other reasons for library unit renaming are

- Essentially all hierarchical naming systems provide some kind of "alias" capability, because of the advantages of allowing the same entity to appear in multiple places within the hierarchy (such as symbolic links in Unix file systems);

- Helper packages that use unchecked conversion to overcome the restrictions on private types may already exist; through the use of library unit renaming, such helper packages can remain available by their current root library unit names, while moving their actual definition into the appropriate place in the library unit hierarchy reflecting their access patterns;

- It is a common practice to establish standard package renaming conventions throughout a project; without library-level renaming, each unit that withs a package must include a copy of the appropriate renaming declaration for the package; this introduces the possibility of nonconformities, and, if done in the specification, overrides the useful lack of transitivity provided for names introduced by with clauses; the program library manager keeps a central record of the project-wide renaming declarations;

- When integrating independently developed libraries into a single library, the hierarchical name space may be used to avoid collisions, and renaming may be used to selectively "export" units from their local name spaces into the root name space for a given user's library;

- Given a large program library, it may be useful to have a flexible subsystem that includes withs for a certain set of units, and then use renaming to select the particular implementations of those units to be used in the particular subsystem configuration appropriate at a given time;

- Renaming can be used to hide from clients the parent child relationship between two packages. Withing a child via a root-level renaming does not give visibility of the parent.

- Renaming is used in the predefined library so that compatibility is preserved for Ada 83 programs. Thus `Ada.Text_IO` is renamed as `Text_IO` as mentioned in II.13.

Some of the capabilities provided by library unit renaming are currently supported by some proprietary program library managers. However, by standardizing these renaming capabilities, very large systems can be built and maintained and rehosted without becoming overly dependent on non-standard program library features.

10.1.3 Children of Generic Packages

As mentioned in II.8, a child may also be generic. However there are a number of important rules to be remembered.

Children of a nongeneric unit may be generic or not but children of a generic unit must always be generic. One of the main problems to be solved in the design of the interaction between genericity and children is the impact of new children on existing instantiations. One possibility would be to say that adding a new child automatically added further corresponding children to all the existing instantiations; this would undoubtedly lead to many surprises and would typically not be what was required since at the time of instantiation the children did not exist and presumably the existing instantiation met the requirements at the time. On the other hand one might decide that existing instantiations did not become extended; however, this would sometimes not be what was wanted either.

Clearly a means is required to enable the user to specify just which units of the hierarchy are to be instantiated. Insisting that all children of a generic unit are themselves generic makes this particularly straightforward and natural; the user just instantiates the units required. The existence of nongeneric children would be a problem because there would not be a natural means to indicate that they were required. One consequence is that it is very common for generic children not to have any generic parameters of their own. See for example the package `Sets` in 3.7.1.

So a typical pattern is

```
generic
   type T is private;
package Parent is
   ...
end Parent;

generic
package Parent.Child is
   ...
end Parent.Child;
```

Since the child has visibility of the formal parameters of its parent it is necessary that the instantiation of the child also has visibility of the corresponding actual parameter of the instantiation of the parent. There are two situations to be considered, instantiation within the parent hierarchy and instantiation outside the hierarchy.

Instantiation inside the parent hierarchy poses no problem since the instantiation has visibility of the parent's formal parameters in the usual way.

Instantiation outside requires that the actual parameter corresponding to the formal parameter of the parent is correspondingly visible to the instantiation of the child. This is assured by requiring that the child is instantiated using the name of the instance of the parent; a with clause for the generic child is necessary in order for the child to be visible. So we might write

```
with Parent;
package Parent_Instance is new Parent(T => Integer);

with Parent.Child;
package Child_Instance is new Parent_Instance.Child;
```

In a sense the with clause for the child makes the generic child visible in every instantiation of the parent and so we can then instantiate it in the usual way.

Note carefully that the instantiations need not be at the library level. An earlier version of Ada 95 did require all instantiations to be at the library level but this was very restrictive for many applications. Of course if we do make the instantiations at the libraray level then the instantiations can themselves form a child hierarchy. However it will be necessary for the child names to be different to those in the generic hierarchy. So we might have

```
with Parent.Child;
package Parent_Instance.Child_Instance is new Parent_Instance.Child;
```

Finally note that there are no restrictions on the instantiation of a generic child of a non-generic parent.

10.1.4 Examples of Interfaces

Programs may use child library units to implement several different kinds of structures. Some possibilities which will now be illustrated are

- A child package may be used to create specialized interfaces to, or views of, an abstraction. This allows independent abstractions to be combined into larger subsystems. The bit-vector and list-based sets example below illustrates this.

- The interface to a system may be organized into subsystems if it is too large to be conveniently implemented as a single package, or if clients do not often need all of the facilities that the package provides. See the discussion of CAIS-A below.

- A hierarchical organization of library units may be used to permit vendor extensions. The form of such extensions can distinguish clearly by package names which parts are standard, and which parts are vendor extensions.

- Child library units may be used to define an extensible, reusable, library of components. Users are encouraged to extend or modify the components by adding new children. An example of this was shown in II.7 as part of a windowing system.

Our first example could be part of a transaction control system for a database. It shows how to use child library units to structure definitions in order to reduce recompilation.

```
package Transaction_Mgt is
   type Transaction is limited private;
   procedure Start_Transaction(Trans: out Transaction);
   procedure Complete_Transaction(Trans: in  Transaction);
   procedure Abort_Transaction(Trans: in  Transaction);
private
   ...
end Transaction_Mgt;

package Transaction_Mgt.Auditing is
   type Transaction_Record is private;
   procedure Log(Rec: in Transaction_Record);
private
   ...
end Transaction_Mgt.Auditing;
```

In the example, some clients require facilities for controlling transactions. Other clients need to be able to log a record of each transaction for auditing purposes. These facilities are logically separate, but transaction records require intimate knowledge of the full structure of a transaction. The solution with child library units is to make a child library unit to support the type `Transaction_Record`. Each unit may be compiled separately. Changes to `Transaction_Mgt.Auditing` do not require recompilation of the parent, `Transaction_Mgt`, nor its clients. But note that withing the child implicitly withs the parent; if this is not desired then, as mentioned above, the child could be renamed thus

```
package Transaction_Auditing renames Transaction_Mgt.Auditing;
```

and then withing `Transaction_Auditing` will not give visibility of `Transaction_Mgt`.

The next example illustrates how child library units can be used to add new interfaces to an existing abstraction. This is useful, for example, when conversion functions are needed between types in independently developed abstractions.

Imagine that two packages exist implementing sets, one using bit vectors, and the other linked lists. The bit vector abstraction may not have an iterator in its interface (a means of taking one element from the set), and hence a function cannot be written to convert from the bit vector set to the linked list set. A child package can be added to the bit vector set package that provides an iterator. A new package could then be written to provide the conversion functions, implemented using the iterator interface.

```
package Bit_Vector_Set is

   type Set is private;
```

```
      function Union(A, B: Set) return Set;
      function Intersect(A, B: Set) return Set;
      function Unit(E: Element) return Set;
      function Empty return Set;
   private
        ...
   end Bit_Vector_Set;

   package List_Set is
      type Set is private;

      function Union(A, B: Set) return Set;
      function Intersect(A, B: Set) return Set;
      function Unit(E: Element) return Set
      function Empty return Set;

      procedure Take(S: in out Set; E: out Element);
      function Is_Empty(S: Set) return Boolean;
   private
      ...
   end List_Set;

   package Bit_Vector_Set.Iterator is

      procedure Take(S: in out Set; E: out Element);
      function Is_Empty(S: Set) return Boolean;

   end Bit_Vector_Set.Iterator;

   with List_Set;
   with Bit_Vector_Set;
   package Set_Conversion is

      function Convert(From: in List_Set.Set)
         return Bit_Vector_Set.Set;
      function Convert(From: in Bit_Vector_Set.Set)
         return List_Set.Set;

   end Set_Conversion;
```

The child package `Bit_Vector_Set.Iterator` adds the two missing subprograms needed in order to iterate over the set. The body of the child package has visibility of the private part of its parent and thus can access the details of the set. This example should be compared with that in 4.4.3 which used class-wide types. Note also that it might be infeasible to modify the body of `Bit_Vector_Set` anyway since it might have been supplied by a third party in encrypted form (such as object code!).

A larger example is provided by CAIS-A [DoD 89b]; this is an interface that provides operating system services in a portable way. The CAIS has a very large specification, with hundreds of packages. The central type, which is manipulated by much of the system, is called `Node_Type`. It is a limited private type defined in a package called `Cais_Definitions`. It is needed throughout the CAIS, but its implementation should be hidden from CAIS application developers. The implementation uses `Unchecked_Conversion` inside packages that manipulate the type. There are a number of subsystems, the largest of which are for manipulating Nodes, Attributes, and I/O. These subsystems also share common types. These common types are implemented using visible types declared in support packages. Only packages in the subsystem are supposed to depend on these support packages.

Using `Unchecked_Conversion` has several disadvantages. First, the replicated type definitions must match exactly, which creates a maintenance problem. Secondly, the compiler must represent each type definition in the same way, which would require a representation clause or good luck. Finally, if the data type is a record, the components may themselves be private types whose definitions may need to be replicated. This propagated need for visibility may cause many or all private type definitions to be replicated in several places.

Child library units could be used to restructure these definitions. The type `Node_Type` might be defined at the root of the hierarchy. Packages that contain types and operations for manipulating Nodes, I/O, and Attributes might be child packages of the root library package. The common types would be private, and the support packages would be child packages of the packages that contain the type definitions.

The Ada binding to POSIX [IEEE 92] is another system with many packages and many types. Some of these types are system independent, and some are explicitly system defined. When designing portable programs it is useful to know when programs depend on system defined definitions, to eliminate such dependencies where possible, and to contain them when they are necessary.

The POSIX-Ada binding attempts to preserve the principle of with list portability: users should be able to determine if a program depends on system defined features by examining the names of the packages in its with clauses. At the same time, the system dependent packages often require visibility on the private types of the standard packages, and in Ada 83 this could only be accomplished by nesting them within the standard packages. Since a nested package never appears in a with clause, the visibility needs of such system dependent packages is in conflict with the principle of with list portability. Child library units offer precisely what is needed to resolve this conflict.

10.1.5 Existing Program Libraries

We anticipate that implementations will continue to support library unit references between program libraries. The hierarchical library unit naming may allow these inter-library references to be handled more naturally within the language, for example, by treating the library units of one program library as child units of an empty package within the referencing library. This would provide automatic name-space separation between the two libraries, since the names of the units of the referenced library would all be prefixed by a hypothetical parent package identifier. This approach would eliminate any need for global uniqueness of library unit names when two or more program libraries are (conceptually) combined in this way. It should also be noted that this provides a good use for private top level library units. Marking a top level library unit as private such as

```
private package PP is ...
```

means that it is not visible to the specifications of public units in the library but only to their bodies. Hence it cannot be accessed directly but only as a slave to other units. Considering the whole library as a hypothetical unit such as `Your_Lib` means that in effect the package becomes

```
private package Your_Lib.PP is ...
```

and then (correctly) cannot be accessed from outside the package `Your_Lib`.

We have avoided specifying standard mechanisms for such inter-program-library references, as implementations vary widely in what they choose to provide in this area. However, the universal availability of hierarchical library unit naming will ensure that a program built out of units from multiple libraries will have a natural and portable representation by using a hierarchical naming approach.

10.1.6 Relationship with Previous Work

We conclude our discussion of hierarchical libraries by mentioning three alternative approaches which were considered and rejected:

Voyeurism. (The term "voyeurism" is due to D. Emery.) This allows packages to request visibility on another package's private declarations via some kind of enhanced context clause (such as **with all** `X;` or **with private** `X;`)

Friends. This allows packages to specify exactly which other packages may have enhanced visibility.

DAG Inheritance. This allows a more general form of hierarchical organization, where packages can, in effect, "nest" within several other packages, without an explicit indication in the parent. This is called "DAG inheritance", since the dependence graph must be a directed acyclic graph.

The voyeur approach has the unsettling characteristic of making the privacy of types a property of other packages' context clauses, instead of a feature of the package that is declaring the type. This inverts the Ada 83 notion of privacy. With child library units, privacy becomes a feature of package hierarchies, which is a generalization of Ada 83's subunit facility. Although many of the same effects can be achieved with either approach, the mechanisms are fundamentally different, and child library units are consistent with this Ada model.

There are serious problems with the **with private** approach — without additional rules, private declarations may be easily reexported to units that do not have a with private clause for the unit. This could happen, for example, via a renaming declaration or deriving from a private type as in the following example

```
package Closed is
   type T is private;
private
   type T is Full_Implementation;
end Closed;

with private Closed;       -- not Ada 95
package Open is
   type D is new Closed.T;
   -- from full declaration of T
end Open;

with Open;
package Surprise is
   type S is new Open.D;
   -- gets full type declaration of T via D
end Surprise;
```

whereby the package `Surprise` gets visibility of the full declaration of `T` via the type `D` declared in the package `Open`.

The real difficulty with the **with private** approach is that dependence can subsequently become diffused.

One of the important advantages of using private types in a package is that the compiler ensures that clients of the package do not become dependent on the details of the type's implementation. This makes it much easier to safely maintain and enhance the implementation of the package, even if the set of clients is large. With child units, if a given client needs more operations on a private type, they must identify those operations and declare them in a public child of the package. When the private type's implementation is revised, only the children of the package need be checked and updated as necessary.

The **with private** approach results in a very different scenario. If a client needs additional access to a private type, they need not separately declare operations on that type in some child. Instead they can simply change their **with** to **with private**. This means that the dependence is now open ended rather than being encapsulated in a child. After a period of use, it is clear that there could be sufficiently widespread and diffuse dependence on the implementation of the private type so that any change will be unacceptably expensive to the clients (since the amount of client code can easily exceed the size of the package), thereby totally defeating the original purpose of private types.

The "friends" approach is used in C++. This solution was considered and was rejected because it does not allow for unplanned extension of a package without requiring modification and recompilation of that package, conflicting with the requirement to reduce recompilation. In the X and CAIS-A examples above we discussed situations where unplanned extension is desirable. It is highly advantageous that Ada 95 support it with this same mechanism.

Furthermore, the "friends" approach inverts Ada's usual client/server philosophy. In general it is not possible to tell at the point of a particular declaration where that declaration will be used. For example, the declaration of a type does not specify the type's users, a task declaration does not specify the task's callers, a library unit's declaration does not specify which other units depend on it, and a generic unit's declaration does not specify where the generic will be instantiated. Allowing a package's declaration to specify which other units may extend that package is inconsistent with this model.

Although on the surface the voyeurism and friends concepts appear simple, the issue of transitivity is problematical. In Ada, context clauses are not transitive. Presumably this would also be the case for voyeur context clauses as well. In that case the meaning of the following program becomes unclear.

```
with private X;            -- not Ada 95
package Y is
   ...
end Y;

with private Y;
with X;
package Q is
   ...
end Q;
```

Here `Q` has visibility to `Y`'s private declarations which in turn may refer to `X`'s private declarations. However, `Q` does not have visibility to `X`'s private declarations through its context clause. Any proposal would have to address in this case whether or not `Q` has visibility to `X`.

The DAG inheritance approach has characteristics of both child library units and voyeurism. If there is more than one private type involved, it is possible that a tree structured hierarchy cannot provide the exact visibility needed. However, the DAG approach is complex and its ramifications far-reaching. We concluded that such a solution was too ambitious for the Ada 9X revision.

10.2 Program Structure

In Ada 83, an executable program consisted of a main subprogram and all other library units reachable from this main subprogram. Execution proceeded by elaborating the entire program, running the main subprogram to completion, waiting for all library-level tasks to complete, and then terminating. Although this model for a program was appropriate in some environments, for many programming environments, a much more dynamic and distributed model is preferable.

In Ada 95, a program may be formed from a cooperating set of partitions. The core language simply says that partitions elaborate independently, communicate, and then terminate in some implementation-defined manner. Each partition has its own environment task to act as the thread of control for library-level elaboration, and to act as the master for library-level tasks.

The description in the core language is kept purposefully non-specific to allow for many different approaches to dynamic and distributed program construction. However, the Distributed Systems annex describes additional standard pragmas and attributes which form the basis for a standard, portable approach to distribution.

10.3 Elaboration

Because Ada allows essentially arbitrary code to execute during the elaboration of library units, it is difficult for the user to ensure that no subprogram is called before it is elaborated. Ada 83 required that access before elaboration be detected, and `Program_Error` raised. This could incur significant overhead at run-time.

Ada 95 addresses both the problem of controlling library unit elaboration order, and the run-time overhead of access-before-elaboration checks.

In Ada 95 the `Elaborate` pragma is effectively replaced by the transitive `Elaborate_All` pragma. `Elaborate_All` on a library unit causes not only the body of that library unit to be elaborated, but also causes the bodies of the library units reachable from that library unit's body to be elaborated. This ensures that any call performed during elaboration on a subprogram defined in the unit to which pragma `Elaborate_All` applies, will not result in an access-before-elaboration error.

The pragma `Elaborate_Body` in a package specifies that the body of the package must be elaborated immediately after its declaration. No intervening elaborations are permitted. This allows the compiler to know whether or not any elaboration-time code exists between a visible subprogram declaration and its body. If there is no such elaboration-time code, or it can be proved to not call the subprogram, then there is no opportunity for access-before-elaboration, and the check may be completely eliminated for the subprogram. Without this pragma, the compiler must assume that other library units might be elaborated between the elaboration of the library unit's declaration and its body, meaning that the check in a visible subprogram cannot be removed by the compiler.

Ada 95 also contains two further pragmas concerned with elaboration. The pragma `Pure` in a package specifies that the package does not have any library-level "state", and may depend only on other pure packages. Pure packages are important for distributed systems.

Finally, the pragma `Preelaborate` indicates that a unit is to be preelaborated; that is elaborated before other units not so indicated; there are restrictions on the actions of a unit which is preelaborated. A unit marked as `Pure` is also preelaborated. The general intent is that certain structures can be set up at link-time (before program execution begins) and then perhaps loaded into RAM.

It is good advice to give all library units one of the pragmas `Pure`, `Preelaborate`, or `Elaborate_Body` in that order of preference, wherever possible. This will ensure the benefits of possible check eliminations as mentioned above.

Further support for preelaboration is described in the Systems Programming annex.

10.4 Library Package Bodies

In Ada 83, if a package does not require a body, but has one nevertheless (perhaps to do some initialization), its body can become out-of-date, and be silently omitted from a subsequent build of an executable program. This can lead to mysterious run-time failures due to the lack of the package body. Ada 95 overcomes this difficulty by allowing a library package to have a body only if a body is required by some language rule. An obvious rule requiring a body is that a subprogram specification in a package specification requires a corresponding subprogram body in the package body. Another rule which is more convenient for ensuring that a body is required in cases where we wish to use the body just for initialization is that the pragma `Elaborate_Body` rather obviously requires a body.

Note that an early version of Ada 95 proposed that a library package always have a body; this was eventually rejected because of the irritating incompatibility problems that would have arisen in moving programs from Ada 83.

10.5 Other Improvements

Child library units take over some of the applications of subunits. However, subunits remain the only way for separately compiling a unit that has visibility to the declarative part of the enclosing unit's body. They are also appropriate for providing bodies for individual units that may be undergoing more active development or maintenance than surrounding units.

To simplify the use of subunits, Ada 95 eliminates the requirement on uniqueness of their simple name within an enclosing library unit. Only the expanded name need be unique. Thus subunits `P.Q.T` and `P.S.T` where `P` is the library package name are allowed in Ada 95 whereas this was forbidden in Ada 83. This is in line with the rules for naming child units.

10.6 Requirements Summary

The requirements includes two study topics

S4.3-A(1) — Reducing the Need for Recompilation

S4.3-C(1) — Enhanced Library Support

which are specifically addressed and well satisfied by the hierarchical library mechanism. In addition the further study topic

S4.3-B(1) — Programming by Specialization/Extension

is also addressed by the hierarchical library in conjunction with type extension.

The requirements

R8.1-A(1) — Facilitating Software Distribution

R8.2-A(1) — Dynamic Reconfiguration

are addressed by the concept of partitions described in 10.2 and elaboration discussed in 10.3. However, this is really the domain of the Distributed Systems annex and the reader is thus referred to Part Three of this rationale for further details.

11 Exceptions

The changes to exception handling from Ada 83 are quite small. The main changes are

- The exception `Numeric_Error` is now a renaming of `Constraint_Error` and is also obsolescent.
- The notion of an exception occurrence is introduced. This refers to an instance of raising an exception. The package `Ada.Exceptions` contains procedures providing additional information regarding an exception occurrence.
- The interaction between exceptions and optimization is clarified.
- A minor improvement is that an accept statement may now directly have an exception handler.

11.1 Numeric Error

Those familiar with the early use of Ada 83 will recall that there was considerable confusion between `Numeric_Error` and `Constraint_Error` in a number of corner situations. As a consequence the ARG ruled that implementations should raise `Constraint_Error` in all situations for which `Numeric_Error` was originally intended. This was a non-binding interpretation with the intention of making it binding in due course.

Essentially all implementations of Ada 83 now raise `Constraint_Error` although for historic reasons some programs may contain dormant handlers for `Numeric_Error`.

The development of Ada 95 provided an opportunity to remove this historic wart once and for all. It was thus proposed that `Numeric_Error` be completely removed. However, many reviewers pointed out that those programs which had conformed to the advice of AI-387 by consistently writing

```
when Constraint_Error | Numeric_Error =>
```

would then become illegal. Accordingly, in Ada 95, `Numeric_Error` is simply a renaming of `Constraint_Error`. Such a change alone would still have made the above illegal because, in Ada 83, all the exceptions in a handler had to be distinct; a supplementary change is thus that an exception may appear more than once in a handler in Ada 95.

Allowing multiple instances of an exception in a given handler has benefits in other areas. It now allows sequences such as

```
when Text_IO.Data_Error | Integer_IO.Data_Error =>
```

where there may be documentation advantages in revealing the potential causes of the exception.

Of course if the user had deliberately relied upon a distinction between `Numeric_Error` and `Constraint_Error` then the program will now become incorrect. It may be simply incompatible but may also be inconsistent if the handlers are in different frames. For a more detailed discussion see A-4.5. Despite this possibility it was concluded that the perhaps safer alternative of completely removing Numeric_Error was not appropriate for this revision although it should be reconsidered at the next revision.

11.2 Exception Occurrences

It is important in many programs to be able to recover from all exceptions and to continue processing in some way. All exceptions including unexpected exceptions can be caught by an **others** handler. Unfortunately, however, Ada 83 provided no way of identifying the particular exception and it was thus not possible to log the details or take specific appropriate action.

This is overcome in Ada 95 by the introduction of the concept of an exception occurrence and a number of subprograms to access information regarding the occurrence. The type Exception_Occurrence and the subprograms are declared in the package Ada.Exceptions. The user can then declare a choice parameter in a handler through which the particular occurrence can be identified. For example a fragment of a continuous embedded system might take the form

```
with Ada.Exceptions;
task body Control is
...
begin
   loop
      begin
         ... -- main algorithm
      exception
         when Error: others =>
            -- unhandled exception; log it
            Log("Unknown error in task Control"
                  &
                  Ada.Exceptions.Exception_Information(Error));
            -- reset data structures as necessary
      end;
      -- loop around to restart the task
   end loop;
end Control;
```

The choice parameter Error is a constant of the type Exception_Occurrence. The function Exception_Information returns a printable string describing the exception and details of the cause of the occurrence. The actual details depend on the implementation.

Two other functions in Ada.Exceptions are Exception_Name and Exception_Message. Exception_Name just returns the name of the exception (the expanded name) and Exception_Message returns a one-liner giving further details (it excludes the name). Thus Exception_Name, Exception_Message and Exception_Information provide a hierarchy of strings appropriate to different requirements.

The purpose of the three functions is to provide information suitable for output and subsequent analysis in a standard way. Although the details of the strings will depend upon the implementation nevertheless they should be appropriate for analysis on that system.

Exception occurrences can be saved for later analysis by the two subprograms Save_Occurrence. Note that the type Exception_Occurrence is limited; using subprograms rather than allowing the user to save values through assignment gives better control over the use of storage for saved exception occurrences (which could of course be large since they may contain full trace back information). The procedure Save_Occurrence may truncate the message to 200 characters whereas the function Save_Occurrence (which returns an access value) is not permitted to truncate the message. (Note that 200 corresponds to the minimum size of line length required to be supported, see 2.2.)

An occurrence may be reraised by calling the procedure Reraise_Occurrence. This is precisely equivalent to reraising an exception by a raise statement without an exception name and does not create a new occurrence (thus ensuring that the original cause is not lost). An advantage of Reraise_Occurrence is that it can be used to reraise an occurrence that was stored by Save_Occurrence.

It is possible to attach a specific message to the raising of an exception by the procedure Raise_Exception. The first parameter is a value of the type Exception_Id which identifies the exception; this value can be obtained by applying the attribute Identity to the identifier of the exception. The second parameter is the message (a string) which can then be retrieved by calling Exception_Message. This provides a convenient means of identifying the cause of an exception during program debugging. Consider

```
declare
   O_Rats: exception;
begin
...
   Raise_Exception(O_Rats'Identity, "Hard cheese");
...
   Raise_Exception(O_Rats'Identity, "Rat poison");
...
exception
   when Event: O_Rats =>
      Put("O_Rats raised because of ");
      Put(Exception_Message(Event));
end;
```

Calling Raise_Exception raises the exception O_Rats with the string attached as the message. The second call of Put in the handler will output Hard cheese or Rat poison according to which occurrence of O_Rats was raised. See also the example in 9.6.1.

Note that the system messages do not include the name so user and system messages can be processed in a similar manner without the user having to insert the exception name in the message.

11.3 Exceptions and Optimization

The general objective is to strike a sensible balance between prescribing the language so rigorously that no optimizations are possible (which would make the language uncompetitive) and allowing so much freedom that the language semantics are almost non-existent (which would impact on portability and provability). A progressive approach is required that enables different degrees to be permitted in different circumstances.

Much of the difficulty lies with exceptions and ensuring that they are still raised in the appropriate frame. In particular we have ensured that calls to subprograms in other library units are not disrupted.

For details of the approach taken the reader is referred to [RM95 11.6]. A more detailed discussion of the rationale will be found in [AARM].

11.4 Other Improvements

As mentioned in 1.3, the description of exception handling has been simplified by the introduction of a new syntactic category handled_sequence_of_statements which embraces a sequence of statements plus associated exception handlers and is used for all situations where handlers are allowed.

An incidental minor improvement following from this change to the syntax is that an accept statement can now have a handler without the necessity for an inner block, thus

```
accept E do
    ...
    ...
exception
    ...
end E;
```

A further practical improvement is that the concept of a current error file is introduced. This can conveniently be used to log error messages without cluttering other output. This is discussed in more detail in Part Three.

11.5 Requirements Summary

The specific requirement

R 4.5-A(1) — Accessing an Exception Name

is met by the introduction of exception occurrences.

The requirement

R2.2-C(1) — Minimize Special Case Restrictions

mentions as an example the curious fact that an accept statement cannot have an exception handler. This has been rectified.

12 Generics

There are a number of important improvements and extensions to the generic model in Ada 95. The extensions are mainly concerned with providing appropriate new parameter mechanisms to match the additional functionality provided by tagged and other new types. In addition problems with flaws in the contract model are cured.

The main changes are

- A distinct formal notation `(<>)` is introduced that enables definite and indefinite subtypes to be treated separately. This cures a major flaw in the contract model. The `(<>)` notation is also used in private types to indicate that they have unknown discriminants.
- There are new formal notations for modular and decimal types.
- The rules for access type matching are extended to accommodate the additional forms of access types.
- There is a new formal notation indicating that the actual type must be derived from a given type. Moreover, in both this case and the existing private formal notation, it is possible to indicate that the type must be tagged.
- There is a new formal notation for package parameters. The generic actual parameter must be an instantiation of a given generic package.
- Minor changes are that static subtype matching is now required for array and access types, and that the order of evaluation of generic actual and default parameters is not so rigidly specified.

12.1 The Contract Model

As mentioned in Part One, there are a number of new forms of generic parameter in Ada 95. Some of these are introduced to correspond to new types such as tagged types, modular types and decimal types. In addition there are new forms for derived types in general and for package parameters; these simplify program composition. All these new forms were introduced in Part One and are discussed in detail in the following sections.

As was discussed in II.11, Ada 83 had a serious violation of the contract model because of the lack of distinction between unconstrained and constrained formal parameters.

The exact distinction is between subtypes for which objects can be declared (without giving any constraints directly or from an initialization) and those for which they cannot. The former category covers scalar subtypes such as `Integer`, constrained array and record subtypes and unconstrained record subtypes which have default discriminants. The term definite is introduced for these subtypes.

Ada 95 cures this flaw in the contract model by requiring that the formal parameter include an unknown discriminant part `(<>)` if an indefinite subtype is to be allowed as actual parameter. In this case the body cannot use the subtype in a context requiring a definite subtype.

On the other hand the existing notation without `(<>)` now indicates that the actual parameter must be definite.

The two notations are illustrated by the following example

```
generic
   type Key(<>) is private;
   type Item    is private;
package Keyed_Index is ... end;
```

The subtype `String`, because it is an unconstrained array subtype, could be associated with `Key`, but not with `Item`. Within the generic, `Key` must not be used to declare a component or uninitialized object.

This is an incompatibility as mentioned in I-4.4 but straightforward to detect and fix. If existing instantiations fail to compile under Ada 95, then the generic unit must be modified to specify that the relevant generic formal allows indefinite subtypes.

This new distinction between definite and indefinite parameters eliminates the primary source of situations in Ada 83 where an otherwise legal instantiation is made illegal by a particular usage pattern of a formal type within the body of the generic unit. In other words this distinction eliminates the major gap in the generic contract model of Ada 83.

Having plugged the major gap in the Ada 83 generic contract model, Ada 95 goes further and ensures that the legality of an instantiation never depends on the parameter usage patterns present in the generic body.

This is achieved in various ways. We have just seen how the addition of further information in the formal parameter enables a satisfactory distinction between usage patterns to be made in the case of definite and indefinite subtypes.

However, it is impracticable to impose all pattern matching requirements through the parameter matching rules. Another approach is to impose certain restrictions in the generic body which in essence assume the "worst" regarding the possible instantiations. An example is that if the generic parameter is nonlimited then all the components in an extension of it also have to be nonlimited. This rule is checked in the instance. For further details of this and other ways in which the contract is ensured see [RM95 12.3]

The general principle is to assume the "best" in the generic specification and then to check the assumptions again at the instantiation, and to assume the "worst" in the body so that legality does not depend upon the instantiation. This of course means that full freedom is not possible in the body but the constraints will not generally be found irksome. A common workaround is to move material from the generic body into the private part.

In conclusion, a tight contract model has several desirable properties. It allows implementations to share generics more easily, it leads to the early detection of programming errors, and it eliminates the need to recheck all existing instantiations when a new body is compiled. Ada 95 strengthens the contract model by requiring that the specification of a generic formal private type indicate whether a corresponding actual may be an unconstrained composite subtype. This simplifies the checking required when a new generic body is compiled, since its legality will not depend on the nature of the existing instantiations.

As pointed out in [DoD 90] in the discussion of Study Topic S4.4-B(2), both tight and loose contract models are desirable, each for its own reasons. This tension has been resolved in Ada 95, by specifying that certain checks in the generic specification are only performed at instantiation time.

Our general goal has been to aim towards the ideal situation whereby within the body of the generic, *all* checks are performed when the generic body is compiled, and these checks fail if *any* possible instantiation could fail the checks. This goal has generally been achieved, (although some errors in the instance are detected at runtime; an example is the use of the `Access` attribute, see 12.3). Ada 95 thus achieves the prime goals associated with a tight contract model, and yet still provides the flexibility required to use generics to their best advantage.

Improving the contract model eases the problems of code sharing for those implementations that use this technique. However, it should be noted that many of the applications of generics where code sharing seemed important can now be done using other techniques such as access to subprogram parameters and tagged types. Moreover, we have not provided an explicit pragma as suggested by the Requirements to control whether code sharing should be used or not since an implementation can use the pragma `Optimize` as suggested in [AARM 2.8(27)].

12.2 Numeric Types

Additional formal types are provided corresponding to unsigned integer types (modular types) and to decimal fixed point types as already mention in 3.3.

The modular types form a subclass of the integer types. They have additional operations such as the logical operations that do not apply to all integer types. As a consequence a signed integer type cannot match a formal modular type. On the other hand modular types behave differently to signed integer types with regard to overflow since by definition they wrap around. And so a modular type cannot match a formal signed integer type.

Similarly the decimal types form a subclass of the fixed point types. Again they are a distinct subclass to the ordinary fixed point types and one cannot match the other. The reason is that an implementation is allowed to use a significantly different representation (such as packed decimal) for decimal types as opposed to ordinary fixed types; it would impose unacceptable inefficiencies on implementations using shared generic bodies to accommodate both kinds of actual via one kind of formal.

12.3 Access Types

Access types are considerably extended in Ada 95 as discussed in 3.7. They may access general objects not created by allocators; they may be marked as constant and there are also access to subprogram types. Accordingly, the matching rules for generic access parameters are adapted to allow for the proper matching of these other forms.

For example if the formal type is

```
type A is access constant T;
```

then the actual type must also be a general access type with the modifier **`constant`**. Similarly, if the formal type is

```
type A is access all T;
```

then the actual type must also be a general access type (with **`all`** but not **`constant`**) that has the type `T` as its accessed type.

In the case of access to subprogram types, the profiles of the formal and actual types have to be mode conformant (see 6.2). This is the same category of conformance as for renaming of subprograms (not renaming bodies) and thus naturally continues the general model that generic parameter matching is renaming.

Note that there are restrictions on the use of the `Access` attribute in a generic body; these are different for access to subprogram and access to object types. The objective is of course to ensure that an access cannot be created to an entity at an inner level.

In the case of an access to subprogram type, the access attribute is not allowed to be applied to a subprogram in a generic body if the access type is external to the generic unit because of worst case considerations. A possible workaround is to move the declaration of the subprogram `P` to the private part and to declare a constant in the private part thus

```
P_Access: constant Global_Access_Type := P'Access;
```

and this will then be checked in the instance of the specification.

In the case of access to object types a different approach is taken. The access attribute is allowed in a generic body but the check occurs dynamically if the access type is external to the body. This check is therefore similar to that for anonymous access parameters and `Program_Error` is raised if it fails.

The different approach relates to the fact that anonymous access types are not allowed for subprogram parameters as discussed in 3.7.2. Also the workaround applicable to access to subprogam types of moving the use of `Access` to the specification cannot usually be applied in the case of access to object types.

12.4 Derived Types

The class-wide programming features of Ada 95 reduce the need to use generics to deal with different types derived from the same root type. However, class-wide programming does not address the important capability, only provided by generics, of defining a new data structure that is parameterized by one or more component types. It is instructive to note that the object oriented programming languages C++, Eiffel, and Modula-3 all include some sort of generic or template mechanism in their latest incarnations.

A new kind of generic formal parameter is provided for derived types. As mentioned above, we see an important role for generics in the definition of new data structures parameterized by one or more component types. For linked data structures, it is often necessary to take advantage of the structure of the components to efficiently implement the (composite) data structure. By using a generic formal derived type, the implementation of the generic can take advantage of the structure and operations of the ancestor type specified for the formal derived type definition.

In the remainder of this section we consider formal (untagged) derived types; tagged types are considered in the next section.

The new notation is

```
type T is new S;
```

which indicates that the actual type must be derived directly or indirectly from `S`.

For a generic formal derived type, the primitive operations available on the type in the generic are determined by the specified ancestor type. Analogous to the rule for formal numeric types, the primitive operations available on an untagged formal derived type use the ancestor operation implementations, even if they have been overridden or hidden for the actual type. This rule is necessary for untagged types, because there is no limitation on the kinds of alterations made to the subtype or mode of the formal parameters when overriding a subprogram inherited by derivation. This contrast strongly with tagged types where the whole essence of the concept is to use replaced operations as described in the next section.

Generic formal derived types permit generic units to be parameterized by a user-defined class — the set of all types derived from the parent type specified in the generic parameter declaration. Within the generic template, the operations of the specified parent type are available. This provides support for user-defined classes that is comparable to that available for language-defined classes, such as discrete, integer, fixed and float.

In a sense therefore the formal parameter notation

```
type T is range <>;
```

is approximately equivalent to

```
type T is new root_integer;
```

although we cannot actually write the latter.

One use of generic formal derived types is to parameterize a generic with a record type but without having to introduce a specific notation for formal record types which would be unwieldy.

The following example is a generic package for providing I/O for types in a user-defined rational class.

```
package Rational_Arithmetic is
   -- this package defines a rational number type

   type Rational is private;

   function "+" (Left, Right: Rational) return Rational;
   ...
end Rational_Arithmetic;

with Rational_Arithmetic; use Rational_Arithmetic;
with Text_IO;
generic
   -- this package provides I/O for any type derived from Rational
   type Num is new Rational;
package Rational_IO is
   procedure Get(File: in Text_IO.File_Type;
                 Item: out Num;
                 Width: in Text_IO.Field := 0);

   procedure Put(File: in Text_IO.File_Type;
                 Item: in Num;
                 Fore: in Text_IO.Field;
                 Aft: in Text_IO.Field;
                 Exp: in Text_IO.Field);
end Rational_IO;
```

The generic formal parameter `Num` will only match `Rational` and its derivatives. Since `Rational` and its derivatives all share the primitive operations of the `Rational` type, those operations are available within `Rational_IO` for implementing the `Get` and `Put` subprograms.

12.5 Tagged Types

Other forms of formal parameters apply to tagged types. Thus

```
type T is tagged private;
```

which simply indicates that the actual type can be any tagged type, and

```
type T is new S with private;
```

which indicates that the actual type can be any extension of the type `S` (or the type `S` itself).

This last form is very important. A form of multiple inheritance is obtained by defining a generic package that extends a formal type with additional components and operations (see 4.6.2). Because type extension is only permitted for tagged types, allowing the reserved word tagged in a generic formal private type declaration makes it clear in the parameter specification that extension might be performed. But note that it is possible to declare a type extension (of a parameter) only in the generic specification; it is not allowed in the generic body. However, as illustrated in 4.6.2 this fits in with typical patterns of use.

The above restriction is an interesting example of the best and worst case contract principle. The underlying rule that we must not violate is that a type extension must be accessible from the parent type declaration as discussed in 4.3. It is thus necessary that an extension in any instantiation also satisfies this rule. In the case of the specification we assume the best and allow an extension. At the point of the instantiation the resulting specification is checked to ensure that the extension does not violate the rule. In the case of the body the general contract principle is that the body must work for any instantiation and accordingly it is not permitted to allow an error to be discovered in the body for a particular instantiation. Thus we assume the worst and forbid any extension since the instantiation might be at a deeper level at which an extension would violate the accessibility rule. This restriction may seem a burden but a commonly applicable workaround is simply to move the type extension and its operations to the private part. An example where this would be necessary is discussed in 4.4.4.

For tagged types, the primitive operations use the implementations defined for the actual type, though this is expressed for consistency in terms of the normal dispatching behavior of the operations of the parent type. For a tagged type it is possible to use the overriding definitions, because these overriding operations must be subtype conformant with the inherited one.

A further refinement is that the formal type can be declared as abstract. In such a case the actual type can also be abstract but it need not. If the formal is abstract then no objects can be declared.

The parameter matching rules are designed to ensure that abstract subprograms are never called. If a type is abstract it does not follow that all its primitive subprograms are abstract. Non-dispatching calls are allowed in the generic unit on only those primitive operations of the formal type which are not abstract. In order to ensure that any instantiation still correctly works it is necessary that the corresponding primitive operations of the actual type are also not abstract. Consider again the package `P` in 7.4.

```
generic
   type Parent is abstract new Limited_Controlled with private;
package P is
   type T is new Parent with private;
   ...
private
   type T is new Parent with
      record
         -- additional components
      end record;
   procedure Finalize(Object: in out T);
end P;
```

then although `Limited_Controlled` is abstract, its primitive operations such as `Finalize` are not abstract and thus calls on `Finalize` are allowed in the body. For this to always work it is essential that the actual type has not replaced the inherited procedure `Finalize` by an abstract one [RM95 3.9.3]. The following is thus illegal

```
type Nasty is abstract new Limited_Controlled with null record;
procedure Finalize(Object: in out Nasty) is abstract;
...
package Q is new P(Parent => Nasty);  -- illegal
```

Class-wide programming and type extension, in combination with generic units, provides many useful facilities.

Generic units may be parameterized by user-defined classes, allowing abstractions to be built around such classes. In this example, `Any_Account` will be matched by any type derived from `Account_With_Interest`. Within the template, the primitive operations of `Account_With_Interest` are available.

```
generic
   type Account_Type(<>) is new Account_With_Interest with private;
package Set_Of_Accounts is
    procedure Add_New_Account(A: in Account_Type);
    procedure Remove_Account(A: in Account_Type);
```

```
      function Balance_Of_Accounts return Money;
      ... -- other operations (e.g. an iterator)
   end Set_Of_Accounts;
```

This generic package could be instantiated with a specific derivative of `Account_With_Interest`, in which case it would be a homogeneous set of such accounts. Alternatively, the generic could be instantiated with a class-wide type like `Account_With_Interest'Class`, in which case it would allow a heterogeneous set of accounts. The notation `(<>)` specifies that the actual account type may have any number of discriminants, or be a class-wide type (that is, it can be indefinite).

12.6 Package Parameters

The final new kind of generic formal parameter is the formal package. A formal package parameter matches any instance of a specified generic package.

Generic formal packages are appropriate in two different circumstances. In the first circumstance, the generic is defining additional operations, or a new abstraction, in terms of some preexisting abstraction defined by some preexisting generic. This kind of "layering" of functionality can be extremely cumbersome if all of the types and operations defined by the preexisting generic must be imported into the new generic. The generic formal package provides a direct way to import all of the types and operations defined in an instance of the preexisting generic.

In other words, generic formal packages allow generics to be parameterized by other generics, which allows for safer and simpler composition of generic abstractions. In particular it allows for one generic to easily extend the abstraction provided by another generic, without requiring the programmer to enumerate all the operations of the first in the formal part of the second. A simple example of the use of this technique was illustrated by the package `Generic_Complex_Vectors` in II.11.

In more elaborate circumstances, there may need to be several formal packages. It then proves convenient to augment the notation

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

which indicates that the actual parameter corresponding to `P` can be *any* package which has been obtained by instantiating `Q` by the notation

```
   with package R is new Q(P1, P2, ...);
```

which indicates that the actual package corresponding to `R` must have been instantiated with the given parameters.

Returning to our example of complex numbers, we can now write a package which exports standard mathematical functions operating on complex numbers and which takes two packages as parameters. One package defines the complex numbers (as in II.11) and the other package is the standard package `Generic_Elementary_Functions` which provides mathematical functions on normal real (that is not complex) numbers. We write

```
   with Generic_Complex_Numbers;
   with Generic_Elementary_Functions;
   generic
      with package Complex_Numbers is
         new Generic_Complex_Numbers(<>);
      with package Elementary_Functions is
         new Generic_Elementary_Functions(Complex_Numbers.Float_Type);
   package Generic_Complex_Functions is
```

```
   use Complex_Numbers;

   function Sqrt(X: Complex) return Complex;
   function Log (X: Complex) return Complex;
   function Exp (X: Complex) return Complex;
   function Sin (X: Complex) return Complex;
   function Cos (X: Complex) return Complex;

end Generic_Complex_Functions;
```

The actual packages must be instantiations of `Generic_Complex_Numbers` and `Generic_Elementary_Functions` respectively. Note that both forms of formal package are used. Any instantiation of `Generic_Complex_Numbers` is allowed but the instantiation of `Generic_Elementary_Functions` must have `Complex_Numbers.Float_Type` as its actual parameter. This ensures that both packages are instantiated with the same floating type.

Note carefully that we are using the formal exported from the first instantiation as the required parameter for the second instantiation. The formal parameters are only accessible in this way when the default form `(<>)` is used. Finally, instantiations might be

```
package Long_Complex_Numbers is
   new Generic_Complex_Numbers(Long_Float);

package Long_Elementary_Functions is
   new Generic_Elementary_Functions(Long_Float);

package Long_Complex_Functions is
   new Generic_Complex_Functions
      (Long_Complex_Numbers, Long_Elementary_Functions);
```

A second circumstance where a generic formal package is appropriate is when the same abstraction is implemented in several different ways. For example, the abstraction of a "mapping" from a key type to a value type is very general, and admits to many different implementation approaches. In most cases, a mapping abstraction can be characterized by a key type, a value type, and operations for adding to the mapping, removing from the mapping, and applying the mapping. This represents a "signature" for the mapping abstraction, and any combination of types and operations that satisfy such a signature syntactically and semantically can be considered a mapping.

A generic package can be used to define a signature, and then a given implementation for the signature is established by instantiating the signature. Once the signature is defined, a generic formal package for this signature can be used in a generic formal part as a short-hand for a type and a set of operations.

We can thus define a generic package `Mapping` that defines the signature of a mapping, and then other generics can be defined with a formal package parameter. The mapping package might be

```
generic
   -- define signature for a Mapping
   type Mapping_Type is limited private;
   type Key is limited private;
   type Value is limited private;
   with procedure Add_Pair(M: in out Mapping_Type;
                           K: in Key;
                           V: in Value);
   with procedure Remove_Pair(M: in out Mapping_Type;
                              K: in Key;
                              V: in Value);
```

```
      with procedure Apply(M: in out Mapping_Type;
                           K: in Key;
                           V: out Value);
   package Mapping is end;
```

We can now define a generic that takes an instance of a `Mapping` as a parameter; for example

```
   generic
      with package Some_Mapping is new Mapping(<>);
      with procedure Do_Something_With_Value(V: Some_Mapping.Value)
   procedure Do_Something_With_Key(K: Some_Mapping.Key);

   procedure Do_Something_With_Key(K: Some_Mapping.Key) is
       V: Some_Mapping.Value;
   begin
      -- translate key to value, and then do something with value
      Some_Mapping.Apply(K, V);
      Do_Something_With_Value(V);
   end Do_Something_With_Key;
```

The reader will note the tedious repetition of `Some_Mapping` in the generic unit. This can be avoided since a use clause is permitted in a generic formal part in Ada 95; the specification can thus be written as

```
   generic
      with package Some_Mapping is new Mapping(<>);
      use Some_Mapping;
      with procedure Do_Something_With_Value(V: Value)
   procedure Do_Something_With_Key(K: Key);
```

with similar changes to the generic body.

Another and more mathematical example is provided by the following which defines the signature of a group.

```
   generic
      type Group_Element is private;
      Identity: in Group_Element;
      with function Op(X, Y: Group_Element) return Group_Element
      with function Inverse(X: Group_Element) return Group_Element;
   package Group_Signature is end;
```

The following generic function applies the group operation to the given group element the specified number of times. If the right operand is negative, the inverse of the result is returned; if it is zero, the identity is returned.

```
   generic
      with package Group is new Group_Signature(<>);
      use Group;
   function Power(X: Group_Element; N: Integer) return Group_Element;

   function Power(X: Group_Element; N: Integer) return Group_Element is
      Result: Group_Element := Identity;
   begin
      for I in 1 .. abs N loop
         Result := Op(Result, X);
      end loop;
      if N < 0 then
```

```
      return Inverse(Result);
   else
      return Result;
   end if;
end Power;
```

The following instantiation ensures that the long complex numbers are a group over addition

```
package Long_Complex_Addition_Group is
   new Group_Signature(Group_Element => Long_Complex.Complex,
                       Identity => (0.0, 0.0);
                       Op => Long_Complex."+";
                       Inverse => Long_Complex."-");
```

and then finally we can instantiate the power function for the long complex addition group as follows

```
function Complex_Multiplication is
   new Power(Long_Complex_Addition_Group);
```

Note that we have assumed that the type `Complex` is not a private type so that the aggregate is available for the identity element.

12.7 Other Improvements

A small change is that the matching of subtypes in array and access types now requires static matching as mentioned in 3.10.

Another minor change is that generic actual parameters are evaluated in an arbitrary order consistent with any dependences whereas in Ada 83 all default parameters were evaluated after all explicit parameters. The relaxation of this ordering requirement brings the rules for generic parameters into line with those for ordinary subprogram parameters.

12.8 Requirements Summary

There were a number of requirements in this area. The study topic

S4.4-A(1) — Generic Formal Parameters

is satisfied by the provision of extra kinds of generic parameters (for derived types and especially package parameters) to enable better abstraction and composition.

The study topic

S4.4-B(2) — Tighten the "Contract Model"

has been met by the provision of separate templates for definite and indefinite types and other modifications to the rules as discussed in 12.1.

The requirement

R4.4-B(1) — Dependence of Instantiations on Bodies

has also been met by the improvements to the contract model.

The requirement

R4.4-C(1) — Generic Code Sharing

is discussed in 12.1 where it is noted that the pragma `Optimize` can be used to control whether sharing is required or not.

13 Representation Issues

This part of the reference manual has always been a pot-pourri of bits and pieces often neglected by users and implementors alike. However, it is an important area especially for embedded systems where tight control of the implementation is required. The changes in Ada 95 are designed to make the vague control promised by Ada 83 into a reality. The main changes are

- The mechanism for specifying representations such as size and alignment is generalized and their meaning is clarified.
- Additional types and operations are provided for address and offset manipulation.
- The `Valid` attribute enables a potentially dubious value (such as might be obtained through calling `Unchecked_Conversion` or interfacing to another language) to be checked for validity.
- Facilities are provided for the more detailed control of heap storage for allocated objects.
- The rules regarding the freezing of representations are properly defined.
- The pragma `Restrictions` is provided for specifying that only a subset of the language is to be used.
- The concept of streams and various stream attributes are introduced.

By their nature, these features of the language concern fine detail for which the reader is referred to [RM95]. In this chapter we will only discuss the broad principles involved. Note that the material on interfacing to other languages is now described in a separate annex of [RM95]; see Chapter B in Part Three. Note also that although the general concept of streams and the stream attributes are defined in section 13 of [RM95], their main application is for input-output and they are therefore discussed in A-1.4.

13.1 Representation of Data

The first point to note is that the notation has been unified so that the attribute form can be used for setting all specifiable attributes. Thus

```
for X'Address use 8#100#;
```

rather than **`for`** `X` **`use at`** `8#100#`, and

```
for R'Alignment use 2;
```

rather than **`at mod`** `2` in a record representation clause. (The old notations are allowed for compatibility although considered obsolete.)

An important reason for the unified notation is that we wish to allow implementations to define additional user-specifiable attributes in a consistent manner. Furthermore the annexes define many additional attributes as discussed in Part Three.

The `Alignment` attribute can be applied to all first subtypes and objects whereas the mod clause only applied to records in Ada 83. An overall rule is that the `Address` of an object must be an integral multiple of its `Alignment`. In the case of internal objects the user must ensure that this is not violated if one or both attributes are specified. In the case of external objects, the attributes may also be specified but then they are more in the nature of an assertion; again it is assumed that the relationship holds.

It is now possible to specify the order of numbering bits. This is particularly important when using record representation clauses to ensure that we know which way round the bits are numbered. For example

```
for R'Bit_Order use Low_Order_First;
```

where `R` is a record type indicates that the first bit (bit 0) is the least significant.

There was much confusion in Ada 83 over the `Size` attribute. This is now clarified and the reader is referred to the discussion in [RM95] for details. An important point is that the `Size` attribute may now be set for individual objects rather than just to types as in Ada 83.

13.2 The Package System

The package `System.Storage_Elements` contains additional types and operations to enable addresses and offsets to be manipulated in a standard manner. The comparison operators are defined for the type `Address` in the package `System` itself whereas other facilities are in the child package `System.Storage_Elements`.

This latter package includes a type `Storage_Offset` and operations to add and subtract such values to and from values of type `Address`. Storage offsets are of course relative whereas addresses are absolute (echoes of duration and time) and so adding a storage offset to an address returns an address and so on.

The generic child package `System.Address_To_Access_Conversions` provides the ability to convert neatly between values of the type `Address` and values of a given general access type; this enables "peeking" and "poking" to be done in a consistent manner.

Finally note that the pragmas `System_Name`, `Storage_Unit` and `Memory_Size` are now obsolete. They were not uniformly implemented in Ada 83 and it was not at all clear what they should mean. For example, in most implementations, it does not make sense to change the number of bits in a storage unit, and even if it did, it would not be sufficient to make only the package `System` obsolete; clearly all generated code depends on this value. Consequently we no longer require implementations to support these pragmas. Of course, implementations that already support them with some particular meaning can continue to do so (as implementation defined pragmas) for upward compatibility. On the other hand the corresponding named numbers in package `System` are quite useful as queries and so remain.

13.3 The Valid Attribute

There are occasions when `Unchecked_Conversion` is very valuable although inherently dangerous. The `Valid` attribute enables the programmer to ensure that the result of an unchecked conversion is at least a valid value for the subtype concerned (even if not what the programmer hoped for). Some risks of catastrophe are thereby avoided.

The [RM95 13.9.2] lists the ways in which invalid data could be obtained. As well as unchecked conversion this includes results obtain through interfacing to another language and uninitialized data. Note that `Valid` is only defined for objects of scalar types.

13.4 Storage Pool Management

For Ada 95, we have provided the user with the ability to override the default algorithms for allocating and deallocating heap storage. This is done by the introduction of the concept of a storage pool which provides the storage for objects created by allocators. Every access to object type is associated with some storage pool which is a pool of memory from which the storage for allocated objects is obtained.

The storage pool for an access type may be shared with other access types. In particular, any derivative of an access type shares the same storage pool as the parent access type. More importantly, an implementation might use a common global heap by default. An allocator for an access type allocates storage from the associated storage pool.

The package `System.Storage_Pools` provides mechanisms for defining a storage pool type as an extension of the abstract type `Root_Storage_Pool`. We can then associate a storage pool with a particular access type by specifying the `Storage_Pool` attribute for the access type. Alternatively, a bounded storage pool may be requested by specifying the `Storage_Size` attribute for an access type, as in Ada 83. In the absence of a specification of either the `Storage_Pool` or `Storage_Size` attribute of an access type, the implementation chooses an appropriate storage pool for the type.

Pool-specific access values never point outside of their storage pool (in the absence of unchecked conversion and the like). On the other hand, general access values may be assigned to point to any aliased object of an appropriate type and scope, either through the use of the `Access` attribute or explicit access type conversion.

The storage pool concept makes explicit the notion of a "heap", and when combined with the ability to specify a `Storage_Pool` object for an access type, gives the user better control over dynamic allocation.

The use of storage pools is illustrated by the following example which shows how an application can use a special allocator algorithm to meet its precise storage requirements. The storage pool associated with the access type supports mark and release operations, allowing rapid reclamation of all storage allocated from the pool during some phase of processing.

```
with System.Storage_Pools;
with System.Storage_Elements; use System;
package Mark_Release_Storage is

   type Mark_Release_Pool(Size: Storage_Elements.Storage_Count) is
         new Storage_Pools.Root_Storage_Pool with private;

   type Pool_Mark is limited private;

   -- now provide the controlled operations
   procedure Initialize(Pool: in out Mark_Release_Pool);

   procedure Finalize(Pool: in out Mark_Release_Pool);

   -- now provide the storage pool operations
   procedure Allocate(
      Pool           : in out Mark_Release_Pool;
      Storage_Address: out Address;
      Size_In_Storage_Elements: in Storage_Elements.Storage_Count;
      Alignment      : in Storage_Elements.Storage_Count);

   procedure Deallocate(
      Pool           : in out Mark_Release_Pool;
      Storage_Address: in Address;
      Size_In_Storage_Elements: in Storage_Elements.Storage_Count;
      Alignment      : in Storage_Elements.Storage_Count);
```

```
      function Storage_Size(Pool: Mark_Release_Pool)
         return Storage_Elements.Storage_Count;

        -- additional subprograms for the Mark_Release_Pool
      procedure Set_Mark(
         Pool: in Mark_Release_Pool;
         Mark: out Pool_Mark);

        -- marks the current state of the pool for later release
      procedure Release_To_Mark(
         Pool: in out Mark_Release_Pool;
         Mark: in Pool_Mark);
        -- frees everything allocated from the Pool since Set_Mark.
        -- all access values designating objects allocated since then
        -- become invalid.

   private
       ...
   end Mark_Release_Storage;
```

This example demonstrates how a package defines a special type of mark/release storage pool, derived from `System.Storage_Pools.Root_Storage_Pool` (see [RM95 13.11]).

Note carefully that the procedures `Allocate` and `Deallocate` are invoked implicitly by the Ada 95 allocator and `Unchecked_Deallocation` facilities in much the same way as `Initialize`, `Adjust` and `Finalize` are implicitly called by the run-time system for controlled types (see 7.4). Moreover, the type `Root_Storage_Pool` is itself a limited controlled type and so the procedures `Initialize` and `Finalize` are provided.

This example includes two additional operations on the storage pool type, which the user can use to set a mark and then later release the pool to a marked state, and thereby reclaim all recently allocated storage. The declaration of `Mark_Release_Pool` indicates that it is also extended with additional private components that would be supplied in the private part.

In order to use the above package we first have to declare a particular pool and then specify it as the pool for the access type concerned. We might write

```
   use Mark_Release_Storage;
   Big_Pool: Mark_Release_Pool(50_000);

   type Some_Type is ...;
   type Some_Access is access Some_Type;
   for Some_Access'Storage_Pool use Big_Pool;
```

This declares the pool `Big_Pool` of the type `Mark_Release_Pool` and then associates it with the access type `Some_Access` by the representation clause. The discriminant of 50,000 acts as an initialization parameter perhaps indicating the total size of the pool. We can then allocate and use objects in the usual way. We can also use the special mark and release capabilities provided by this particular type of pool.

```
   declare
      Mark: Pool_Mark;
      Done: Boolean := False;
   begin
      -- set mark prior to commencing the loop
      Set_Mark(Big_Pool, Mark);
      while not Done loop
         -- each iteration allocates a data structure composed of
```

```
         -- objects of Some_Type, which may be discarded
         -- before the next iteration.
         declare
            X, Y : Some_Access;
         begin
            -- algorithm that allocates various objects from
            -- the mark/release storage pool
            X := new Some_Type;
            ...

            Y := new Some_Type;
            ...
            -- release storage each time through the loop
            Release_To_Mark(Big_Pool, Mark);
         exception
            when others =>
               -- release storage then reraise the exception
               Release_To_Mark(Big_Pool, Mark);
               raise;
         end;
      end loop;
   end;
```

Note carefully that the assignments such as

```
X := new Some_Type;
```

implicitly call the `Allocate` procedure thus

```
Mark_Release_Storage.Allocate(Pool => Big_Pool,...);
```

Any calls on `Unchecked_Deallocation` will similarly result in an implicit call of `Deallocate`.

13.5 Freezing of Representations

Certain uses of an entity or the type of an entity cause it to be frozen; these are situations where the representation has to be known (and if none has been specified the implementation will then choose a representation by default). These uses were called forcing occurrences in Ada 83 (the name has been changed because not all the situations causing freezing are actual occurrences of the name of the entity). The forcing occurrence rules of Ada 83 did not really achieve their objective; sometimes they were too lax and sometimes too rigid; the freezing rules of Ada 95 are intended to more exactly satisfy the objective of identifying when the representation has to be determined.

The situations causing freezing and the operations not allowed on a frozen entity are described in [RM95 13.14]. There seems little point in repeating the discussion here but one point of difference worth emphasizing is that the loophole in deferred constants in Ada 83 which allowed uninitialized access values is now blocked. The new rules were designed to overcome this and similar problems with the Ada 83 rules.

13.5.1 Freezing and Tagged Types

The freezing rules for tagged types are important. The two main ones are that a record extension freezes the parent and a primitive subprogram cannot be declared for a frozen tagged type — this applies to both new ones and overridden ones. Using the illustrative model of the tag and dispatch table in 4.3 this means that the contents of the dispatch table can be determined as soon as the type is frozen.

A consequence of these freezing rules is that we cannot declare further primitive subprograms for a tagged type after a record extension of it has been defined. This was mentioned in II.1 during the discussion of the alert system when we noted the practical benefit of being able to declare a sequence of derived types in one package.

But note that although a record extension freezes the parent type a private extension does not. In the private case the parent type is frozen at the full type declaration (which will be a record extension anyway). So in the following

```
package P is
   type T is tagged ...;
   type NT is new T with private;
   procedure Op(X: T);
private
   type NT is new T with ...;
end P;
```

the partial declaration of `NT` does not freeze `T` and so the further operation `Op` can be added. This operation is also inherited by `NT` although it is not visible for the partial view of `NT` (since its declaration was after that of the partial view); it effectively gets added at the full declaration. So

```
A: NT'Class := ...;
Op(A);
```

is illegal outside `P` but legal in the body of `P`.

Note that we can add an operation, `OpN` for `NT` before the new operation `Op` for `T` thus

```
package P is
   type T is tagged ...;
   type NT is new T with private;
   procedure OpN(X: NT);
   procedure Op(X: T);
private
   type NT is new T with ...;
end P;
```

and in this case, perhaps surprisingly, we have added a new operation for the dispatch table of `NT` before knowing all about the dispatch table of `T` (which of course forms the first part of the table for `NT`). However, the full declaration of `NT` will freeze `T` and prevent further operations being added for `T`.

It is instructive to reconsider the alert system discussed in Part One and to rearrange the declarations to minimize spurious visibility. The details of the various types need not be visible externally (we can imagine that they are initialized by operations in some child package). Moreover, it is only necessary for the procedure `Handle` to be visible externally since `Display`, `Log` and `Set_Alarm` are only called internally from the procedures `Handle`. However, in the case of `Display` it is important that it be a dispatching operation if the redispatching discussed in 4.5 is to be possible. The package could thus be reconstructed as follows

```
with Calendar;
package New_Alert_System is

   type Alert is abstract tagged private;
   procedure Handle(A: in out Alert);

   type Low_Alert is new Alert with private;
   type Medium_Alert is new Alert with private;
```

```
      type High_Alert is new Alert with private;

   private

      type Device is (Teletype, Console, Big_Screen);

      type Alert is tagged
         record
            Time_Of_Arrival: Calendar.Time;
            Message: Text;
         end record;

      procedure Display(A: in Alert; On: in Device);  -- also dispatches
      procedure Log(A: in Alert);

      type Low_Alert is new Alert with null record;

      type Medium_Alert is new Alert with
         record
            Action_Officer: Person;
         end record;

      -- now override inherited operations
      procedure Handle(MA: in out Medium_Alert);
      procedure Display(MA: in Medium_Alert; On: in Device);

      type High_Alert is new Medium_Alert with
         record
            Ring_Alarm_At: Calendar.Time;
         end record;

      procedure Handle(HA: in out High_Alert);
      procedure Display(HA: in High_Alert; On: in Device);
      procedure Set_Alarm(HA: in High_Alert);

   end New_Alert_System;
```

In this formulation all the alerts are private and the visible part consists of a series of private extensions. If the private extensions froze the parent type `Alert` then it would not be possible to add the private dispatching operation `Display` in the private part. The deferral of freezing until the full type declaration is thus important. Note that we have also hidden the fact that the `High_Alert` is actually derived from `Medium_Alert`. Remember that the full type declaration only has to have the type given in the private extension as some ancestor and not necessarily as the immediate parent.

We can now add a child package for the emergency alert as suggested in II.7 and this will enable a new `Display` and `Handle` to be added.

```
   package New_Alert_System.Emergency is
      type Emergency_Alert is new Alert with private;
   private
      type Emergency_Alert is new Alert with
         record
            ...
         end record;

      procedure Handle(EA: in out Emergency_Alert);
      procedure Display(EA: in Emergency_Alert; On: in Device);
```

```
    end New_Alert_System.Emergency;
```

We could make the procedure `Display` visible by declaring it in the visible part; it would still override the inherited version even though the inherited version is private as mentioned in 7.1.1.

13.6 The Pragma Restrictions

There are some application areas where it is useful to impose restrictions on the use of certain features of the language. Thus it might be desirable to know that only certain simple uses of tasking are made in a particular program; this might allow the program to be linked with an especially efficient run-time system for use in a hard real time application. Another area where more severe restrictions are relevant is for safety-critical applications where it is required that application programs are written using only simple parts of the language so that they are more amenable to mathematical proof. Restrictions on the use of the language may be imposed by the pragma `Restrictions`. The possible arguments to this pragma are defined in the Real-Time Systems and Safety and Security annexes [RM95 D7, H3].

13.7 Requirements Summary

The requirement

R6.2-A(1) — Data Interoperability

is partially met by the provision of better control over representations such as the alignment of objects.

The study topic

S6.4-B(1) — Low-Level Pointer Operations

is addressed by the attribute `Unchecked_Access` and address and offset operations in the package `System.Address_To_Access_Conversions`.

The requirement

R4.2-A(1) — Allocation and Reclamation of Storage

is met by the storage pool mechanism described in 13.4.