



INTERNATIONAL STANDARD ISO/IEC 8652:1995
TECHNICAL CORRIGENDUM 1

Published 2001-06-01

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION • МЕЖДУНАРОДНАЯ ОРГАНИЗАЦИЯ ПО СТАНДАРТИЗАЦИИ • ORGANISATION INTERNATIONALE DE NORMALISATION
INTERNATIONAL ELECTROTECHNICAL COMMISSION • МЕЖДУНАРОДНАЯ ЭЛЕКТРОТЕХНИЧЕСКАЯ КОМИССИЯ • COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

Information technology — Programming languages — Ada

TECHNICAL CORRIGENDUM 1

Technologies de l'information — Langues de programmation — Ada

RECTIFICATIF TECHNIQUE 1

Technical Corrigendum 1 to International Standard ISO/IEC 8652:1995 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Introduction

This corrigendum contains corrections to the Ada 95 standard [ISO/IEC 8652:1995].

The corrigendum is organized by sections corresponding to those in the Ada 95 standard. These sections include wording changes to the Ada 95 standard. Clause and subclause headings are given for each clause that contains a wording change. Other clauses are omitted. For each change, a reference to the defect report(s) that prompted the wording change is included in the form [8652/0000]. The defect reports have been developed by the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group to address specific questions about the Ada standard. Refer to the defect reports for details on the issues.

For each change, an *anchor* paragraph from the original Ada 95 standard is given. New or revised text and instructions are given with each change. The anchor paragraph can be replaced or deleted, or text can be inserted before or after it. When a heading immediately precedes the anchor paragraph, any text inserted before the paragraph is intended to appear under the heading.

Typographical conventions:

Instructions about the text changes are in this font. The actual text changes are in the same fonts as the Ada 95 standard - this font for text, this font for syntax, and this font for Ada source code.

Section 1: General

1.2 Normative References

Replace paragraph 8: [8652/0001]

ISO/IEC 10646-1:1993, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*.

by:

ISO/IEC 10646-1:1993, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*, supplemented by Technical Corrigendum 1:1996.

Section 2: Lexical Elements

No changes in this section.

Section 3: Declarations and Types

3.3.1 Object Declarations

Replace paragraph 18: [8652/0002]

3. The object is created, and, if there is not an initialization expression, any per-object expressions (see 3.8) are evaluated and any implicit initial values for the object or for its subcomponents are obtained as determined by the nominal subtype.

by:

3. The object is created, and, if there is not an initialization expression, any per-object constraints (see 3.8) are elaborated and any implicit initial values for the object or for its subcomponents are obtained as determined by the nominal subtype.

3.5.4 Integer Types

Insert after paragraph 27: [8652/0003]

For a one's complement machine, the high bound of the base range of a modular type whose modulus is one less than a power of 2 may be equal to the modulus, rather than one less than the modulus. It is implementation defined for which powers of 2, if any, this permission is exercised.

the new paragraph:

For a one's complement machine, implementations may support non-binary modulus values greater than System.Max_Nonbinary_Modulus. It is implementation defined which specific values greater than System.Max_Nonbinary_Modulus, if any, are supported.

3.5.8 Operations of Floating Point Types

Replace paragraph 2: [8652/0004]

S'Digits S'Digits denotes the requested decimal precision for the subtype S. The value of this attribute is of the type *universal_integer*. The requested decimal precision of the base subtype of a floating point type T is defined to be the largest value of d for which $\text{ceiling}(d * \log(10) / \log(T'Machine_Radix)) + 1 \leq T'Model_Mantissa$.

by:

S'Digits S'Digits denotes the requested decimal precision for the subtype S. The value of this attribute is of the type *universal_integer*. The requested decimal precision of the base subtype of a floating point type T is defined to be the largest value of d for which

$$\text{ceiling}(d * \log(10) / \log(T'Machine_Radix)) + g \leq T'Model_Mantissa$$

where g is 0 if Machine_Radix is a positive power of 10 and 1 otherwise.

3.5.10 Operations of Fixed Point Types

Replace paragraph 2: [8652/0005]

S'Small S'Small denotes the *small* of the type of S. The value of this attribute is of the type *universal_real*. Small may be specified for nonderived fixed point types via an attribute_definition_clause (see 13.3); the expression of such a clause shall be static.

by:

S'Small S'Small denotes the *small* of the type of S. The value of this attribute is of the type *universal_real*. Small may be specified for nonderived ordinary fixed point types via an attribute_definition_clause (see 13.3); the expression of such a clause shall be static.

3.6 Array Types

Replace paragraph 22: [8652/0002]

The elaboration of a `discrete_subtype_definition` creates the discrete subtype, and consists of the elaboration of the `subtype_indication` or the evaluation of the `range`. The elaboration of a `component_definition` in an `array_type_definition` consists of the elaboration of the `subtype_indication`. The elaboration of any `discrete_subtype_definitions` and the elaboration of the `component_definition` are performed in an arbitrary order.

by:

The elaboration of a `discrete_subtype_definition` that does not contain any per-object expressions creates the discrete subtype, and consists of the elaboration of the `subtype_indication` or the evaluation of the `range`. The elaboration of a `discrete_subtype_definition` that contains one or more per-object expressions is defined in 3.8. The elaboration of a `component_definition` in an `array_type_definition` consists of the elaboration of the `subtype_indication`. The elaboration of any `discrete_subtype_definitions` and the elaboration of the `component_definition` are performed in an arbitrary order.

3.6.2 Operations of Array Types

Replace paragraph 2: [8652/0006]

The following attributes are defined for a prefix `A` that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

by:

The following attributes are defined for a prefix `A` that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

3.7 Discriminants

Replace paragraph 8: [8652/0007]

A `known_discriminant_part` is only permitted in a declaration for a composite type that is not an array type (this includes generic formal types); a type declared with a `known_discriminant_part` is called a *discriminated* type, as is a type that inherits (known) discriminants.

by:

A `discriminant_part` is only permitted in a declaration for a composite type that is not an array type (this includes generic formal types). A type declared with a `known_discriminant_part` is called a *discriminated* type, as is a type that inherits (known) discriminants.

3.7.1 Discriminant Constraints

Replace paragraph 7: [8652/0008]

A `discriminant_constraint` is only allowed in a `subtype_indication` whose `subtype_mark` denotes either an unconstrained discriminated subtype, or an unconstrained access subtype whose designated subtype is an unconstrained discriminated subtype.

by:

A `discriminant_constraint` is only allowed in a `subtype_indication` whose `subtype_mark` denotes either an unconstrained discriminated subtype, or an unconstrained access subtype whose designated subtype is an unconstrained discriminated subtype. However, in the case of a general access subtype, a `discriminant_constraint` is illegal if there is a place within the immediate scope of the designated subtype where the designated subtype's view is constrained.

3.8 Record Types

Replace paragraph 5: [8652/0009]

`component_item ::= component_declaration | representation_clause`

by:

`component_item ::= component_declaration | aspect_clause`

Replace paragraph 18: [8652/0002]

Within the definition of a composite type, if a `component_definition` or `discrete_subtype_definition` (see 9.5.2) includes a `name` that denotes a discriminant of the type, or that is an `attribute_reference` whose `prefix` denotes the current instance of the type, the expression containing the `name` is called a *per-object expression*, and the constraint being defined is called a *per-object constraint*. For the elaboration of a `component_definition` or a `component_declaration`, if the constraint of the `subtype_indication` is not a per-object constraint, then the `subtype_indication` is elaborated. On the other hand, if the constraint is a per-object constraint, then the elaboration consists of the evaluation of any included expression that is not part of a per-object expression.

by:

Within the definition of a composite type, if a `component_definition` or `discrete_subtype_definition` (see 9.5.2) includes a `name` that denotes a discriminant of the type, or that is an `attribute_reference` whose `prefix` denotes the current instance of the type, the expression containing the `name` is called a *per-object expression*, and the constraint or range being defined is called a *per-object constraint*. For the elaboration of a `component_definition` or a `component_declaration` or the `discrete_subtype_definition` of an `entry_declaration` for an entry family (see 9.5.2), if the constraint or range of the `subtype_indication` or `discrete_subtype_definition` is not a per-object constraint, then the `subtype_indication` or `discrete_subtype_definition` is elaborated. On the other hand, if the constraint or range is a per-object constraint, then the elaboration consists of the evaluation of any included expression that is not part of a per-object expression. Each such expression is evaluated once unless it is part of a named association in a discriminant constraint, in which case it is evaluated once for each associated discriminant.

When a per-object constraint is elaborated (as part of creating an object), each per-object expression of the constraint is evaluated. For other expressions, the values determined during the elaboration of the `component_definition` or `entry_declaration` are used. Any checks associated with the enclosing `subtype_indication` or `discrete_subtype_definition` are performed, including the subtype compatibility check (see 3.2.2), and the associated subtype is created.

3.9.2 Dispatching Operations of Tagged Types

Replace paragraph 7: [8652/0010]

A `type_conversion` is statically or dynamically tagged according to whether the type determined by the `subtype_mark` is specific or class-wide, respectively. For a controlling operand that is designated by an actual parameter, the controlling operand is statically or dynamically tagged according to whether the designated type of the actual parameter is specific or class-wide, respectively.

by:

A `type_conversion` is statically or dynamically tagged according to whether the type determined by the `subtype_mark` is specific or class-wide, respectively. For an object that is designated by an expression whose expected type is an anonymous access-to-specific tagged type, the object is dynamically tagged if the expression, ignoring enclosing parentheses, is of the form `X'Access`, where `X` is of a class-wide type, or is of the form `new T(...)`, where `T` denotes a class-wide subtype. Otherwise, the object is statically or dynamically tagged according to whether the designated type of the type of the expression is specific or class-wide, respectively.

Replace paragraph 9: [8652/0010]

If the expected type for an expression or `name` is some specific tagged type, then the expression or `name` shall not be dynamically tagged unless it is a controlling operand in a call on a dispatching operation. Similarly, if the expected type for an expression is an anonymous access-to-specific tagged

type, then the expression shall not be of an access-to-class-wide type unless it designates a controlling operand in a call on a dispatching operation.

by:

If the expected type for an expression or name is some specific tagged type, then the expression or name shall not be dynamically tagged unless it is a controlling operand in a call on a dispatching operation. Similarly, if the expected type for an expression is an anonymous access-to-specific tagged type, then the object designated by the expression shall not be dynamically tagged unless it is a controlling operand in a call on a dispatching operation.

Replace paragraph 10: [8652/0011]

In the declaration of a dispatching operation of a tagged type, everywhere a subtype of the tagged type appears as a subtype of the profile (see 6.1), it shall statically match the first subtype of the tagged type. If the dispatching operation overrides an inherited subprogram, it shall be subtype conformant with the inherited subprogram. A dispatching operation shall not be of convention Intrinsic. If a dispatching operation overrides the predefined equals operator, then it shall be of convention Ada (either explicitly or by default — see 6.3.1).

by:

In the declaration of a dispatching operation of a tagged type, everywhere a subtype of the tagged type appears as a subtype of the profile (see 6.1), it shall statically match the first subtype of the tagged type. If the dispatching operation overrides an inherited subprogram, it shall be subtype conformant with the inherited subprogram. The convention of an inherited or overriding dispatching operation is the convention of the corresponding primitive operation of the parent type. An explicitly declared dispatching operation shall not be of convention Intrinsic.

3.10 Access Types

Replace paragraph 7: [8652/0012]

There are two kinds of access types, *access-to-object* types, whose values designate objects, and *access-to-subprogram* types, whose values designate subprograms. Associated with an access-to-object type is a *storage pool*; several access types may share the same storage pool. A storage pool is an area of storage used to hold dynamically allocated objects (called *pool elements*) created by allocators; storage pools are described further in 13.11, “Storage Management”.

by:

There are two kinds of access types, *access-to-object* types, whose values designate objects, and *access-to-subprogram* types, whose values designate subprograms. Associated with an access-to-object type is a *storage pool*; several access types may share the same storage pool. All descendants of an access type share the same storage pool. A storage pool is an area of storage used to hold dynamically allocated objects (called *pool elements*) created by allocators; storage pools are described further in 13.11, “Storage Management”.

Replace paragraph 14: [8652/0013]

All subtypes of an access-to-subprogram type are constrained. The first subtype of a type defined by an *access_type_definition* or an *access_to_object_definition* is unconstrained if the designated subtype is an unconstrained array or discriminated type; otherwise it is constrained.

by:

All subtypes of an access-to-subprogram type are constrained. The first subtype of a type defined by an *access_definition* or an *access_to_object_definition* is unconstrained if the designated subtype is an unconstrained array or discriminated subtype; otherwise it is constrained.

3.10.2 Operations of Access Types

Replace paragraph 24: [8652/0010]

X'Access

X'Access yields an access value that designates the object denoted by X. The type of X'Access is an access-to-object type, as determined by the expected type. The expected type shall be a general access type. X shall denote an aliased view of an object, including possibly the current instance (see 8.6) of a limited type within its definition, or a formal parameter or generic formal object of a tagged type. The view denoted by the prefix X shall satisfy the following additional requirements, presuming the expected type for X'Access is the general access type A:

by:

X'Access

X'Access yields an access value that designates the object denoted by X. The type of X'Access is an access-to-object type, as determined by the expected type. The expected type shall be a general access type. X shall denote an aliased view of an object, including possibly the current instance (see 8.6) of a limited type within its definition, or a formal parameter or generic formal object of a tagged type. The view denoted by the prefix X shall satisfy the following additional requirements, presuming the expected type for X'Access is the general access type A, with designated type D:

Replace paragraph 27: [8652/0010]

- If the designated type of A is tagged, then the type of the view shall be covered by the designated type; if A's designated type is not tagged, then the type of the view shall be the same, and either A's designated subtype shall statically match the nominal subtype of the view, or the designated subtype shall be discriminated and unconstrained;

by:

- If A is a named access type and D is a tagged type, then the type of the view shall be covered by D; if A is anonymous and D is tagged, then the type of the view shall be either D'Class or a type covered by D; if D is untagged, then the type of the view shall be D, and A's designated subtype shall either statically match the nominal subtype of the view or be discriminated and unconstrained;

3.11 Declarative Parts

Replace paragraph 4: [8652/0009]

```
basic_declarative_item ::=
    basic_declaration | representation_clause | use_clause
```

by:

```
basic_declarative_item ::=
    basic_declaration | aspect_clause | use_clause
```

Replace paragraph 10: [8652/0014]

- For a call to a (non-protected) subprogram that has an explicit body, a check is made that the subprogram_body is already elaborated. This check and the evaluations of any actual parameters of the call are done in an arbitrary order.

by:

- For a call to a (non-protected) subprogram that has an explicit body, a check is made that the body is already elaborated. This check and the evaluations of any actual parameters of the call are done in an arbitrary order.

3.11.1 Completions of Declarations

Replace paragraph 1: [8652/0014]

Declarations sometimes come in two parts. A declaration that requires a second part is said to *require completion*. The second part is called the *completion* of the declaration (and of the entity declared), and is either another declaration, a body, or a pragma.

by:

Declarations sometimes come in two parts. A declaration that requires a second part is said to *require completion*. The second part is called the *completion* of the declaration (and of the entity declared), and is either another declaration, a body, or a pragma. A *body* is a *body*, an *entry_body*, or a *renaming-as-body* (see 8.5.4).

Section 4: Names and Expressions

4.1.4 Attributes

Replace paragraph 12: [8652/0015]

An implementation may provide implementation-defined attributes; the identifier for an implementation-defined attribute shall differ from those of the language-defined attributes.

by:

An implementation may provide implementation-defined attributes; the identifier for an implementation-defined attribute shall differ from those of the language-defined attributes unless supplied for compatibility with a previous edition of this International Standard.

4.5.2 Relational Operators and Membership Tests

Insert after paragraph 24: [8652/0016]

- Otherwise, the result is defined in terms of the primitive equals operator for any matching tagged components, and the predefined equals for any matching untagged components.

the new paragraph:

For any composite type, the order in which "=" is called for components is unspecified. Furthermore, if the result can be determined before calling "=" on some components, it is unspecified whether "=" is called on those components.

Insert after paragraph 32: [8652/0016]

A membership test using **not in** gives the complementary result to the corresponding membership test using **in**.

the new paragraph:

Implementation Requirements

For all nonlimited types declared in language-defined packages, the "=" and "/=" operators of the type shall behave as if they were the predefined equality operators for the purposes of the equality of composite types and generic formal types.

4.6 Type Conversions

Replace paragraph 5: [8652/0017]

A *type_conversion* whose operand is the name of an object is called a *view conversion* if its target type is tagged, or if it appears as an actual parameter of mode **out** or **in out**; other *type_conversions* are called *value conversions*.

by:

A *type_conversion* whose operand is the name of an object is called a *view conversion* if both its target type and operand type are tagged, or if it appears as an actual parameter of mode **out** or **in out**; other *type_conversions* are called *value conversions*.

Replace paragraph 11: [8652/0008]

- Corresponding index types shall be convertible; and

by:

- Corresponding index types shall be convertible;

Replace paragraph 12: [8652/0008]

- The component subtypes shall statically match.

by:

- The component subtypes shall statically match; and
- In a view conversion, the target type and the operand type shall both or neither have aliased components.

Replace paragraph 54: [8652/0017]

- If the target type is composite, the bounds or discriminants (if any) of the view are as defined above for a value conversion; each nondiscriminant component of the view denotes the matching component of the operand object; the subtype of the view is constrained if either the target subtype or the operand object is constrained, or if the operand type is a descendant of the target type, and has discriminants that were not inherited from the target type;

by:

- If the target type is composite, the bounds or discriminants (if any) of the view are as defined above for a value conversion; each nondiscriminant component of the view denotes the matching component of the operand object; the subtype of the view is constrained if either the target subtype or the operand object is constrained, or if the target subtype is indefinite, or if the operand type is a descendant of the target type and has discriminants that were not inherited from the target type;

4.8 Allocators

Replace paragraph 3: [8652/0010]

The expected type for an allocator shall be a single access-to-object type whose designated type covers the type determined by the `subtype_mark` of the `subtype_indication` or `qualified_expression`.

by:

The expected type for an allocator shall be a single access-to-object type with designated type *D* such that either *D* covers the type determined by the `subtype_mark` of the `subtype_indication` or `qualified_expression`, or the expected type is anonymous and the determined type is *D*'Class.

Replace paragraph 10: [8652/0002]

- If the designated type is composite, an object of the designated type is created with tag, if any, determined by the `subtype_mark` of the `subtype_indication`; any per-object constraints on subcomponents are elaborated and any implicit initial values for the subcomponents of the object are obtained as determined by the `subtype_indication` and assigned to the corresponding subcomponents. A check is made that the value of the object belongs to the designated subtype. `Constraint_Error` is raised if this check fails. This check and the initialization of the object are performed in an arbitrary order.

by:

- If the designated type is composite, an object of the designated type is created with tag, if any, determined by the `subtype_mark` of the `subtype_indication`; any per-object constraints on subcomponents are elaborated (see 3.8) and any implicit initial values for the subcomponents of the object are obtained as determined by the `subtype_indication` and assigned to the corresponding subcomponents. A check is made that the value of the object belongs to the designated subtype. `Constraint_Error` is raised if this check fails. This check and the initialization of the object are performed in an arbitrary order.

Section 5: Statements

No changes in this section.

Section 6: Subprograms

6.3.1 Conformance Rules

Replace paragraph 2: [8652/0011]

As explained in B.1, “‘Interfacing Pragmas’”, a *convention* can be specified for an entity. For a callable entity or access-to-subprogram type, the convention is called the *calling convention*. The following conventions are defined by the language:

by:

As explained in B.1, “‘Interfacing Pragmas’”, a *convention* can be specified for an entity. Unless this International Standard states otherwise, the default convention of an entity is Ada. For a callable entity or access-to-subprogram type, the convention is called the *calling convention*. The following calling conventions are defined by the language:

Insert after paragraph 13: [8652/0011]

- The default calling convention is *entry* for an entry.

the new paragraph:

- If not specified above as Intrinsic, the calling convention for any inherited or overriding dispatching operation of a tagged type is that of the corresponding subprogram of the parent type. The default calling convention for a new dispatching operation of a tagged type is the convention of the type.

Insert after paragraph 21: [8652/0018]

- each `direct_name`, `character_literal`, and `selector_name` that is not part of the prefix of an expanded name in one denotes the same declaration as the corresponding `direct_name`, `character_literal`, or `selector_name` in the other; and

the new paragraph:

- each `attribute_designator` in one must be the same as the corresponding `attribute_designator` in the other; and

Section 7: Packages

7.3.1 Private Operations

Replace paragraph 3: [8652/0019]

For a composite type, the characteristics (see 7.3) of the type are determined in part by the characteristics of its component types. At the place where the composite type is declared, the only characteristics of component types used are those characteristics visible at that place. If later within the immediate scope of the composite type additional characteristics become visible for a component type, then any corresponding characteristics become visible for the composite type. Any additional predefined operators are implicitly declared at that place.

by:

For a composite type, the characteristics (see 7.3) of the type are determined in part by the characteristics of its component types. At the place where the composite type is declared, the only characteristics of component types used are those characteristics visible at that place. If later immediately within the declarative region in which the composite type is declared additional characteristics become visible for a component type, then any corresponding characteristics become visible for the composite type. Any additional predefined operators are implicitly declared at that place.

Replace paragraph 4: [8652/0019]

The corresponding rule applies to a type defined by a `derived_type_definition`, if there is a place within its immediate scope where additional characteristics of its parent type become visible.

by:

The corresponding rule applies to a type defined by a `derived_type_definition`, if there is a place immediately within the declarative region in which the type is declared where additional characteristics of its parent type become visible.

Replace paragraph 5: [8652/0019]

For example, an array type whose component type is limited private becomes nonlimited if the full view of the component type is nonlimited and visible at some later place within the immediate scope of the array type. In such a case, the predefined "=" operator is implicitly declared at that place, and assignment is allowed after that place.

by:

For example, an array type whose component type is limited private becomes nonlimited if the full view of the component type is nonlimited and visible at some later place immediately within the declarative region in which the array type is declared. In such a case, the predefined "=" operator is implicitly declared at that place, and assignment is allowed after that place.

Replace paragraph 6: [8652/0019]

Inherited primitive subprograms follow a different rule. For a `derived_type_definition`, each inherited primitive subprogram is implicitly declared at the earliest place, if any, within the immediate scope of the `type_declaration`, but after the `type_declaration`, where the corresponding declaration from the parent is visible. If there is no such place, then the inherited subprogram is not declared at all. An inherited subprogram that is not declared at all cannot be named in a call and cannot be overridden, but for a tagged type, it is possible to dispatch to it.

by:

Inherited primitive subprograms follow a different rule. For a `derived_type_definition`, each inherited primitive subprogram is implicitly declared at the earliest place, if any, immediately within the declarative region in which the `type_declaration` occurs, but after the `type_declaration`, where the corresponding declaration from the parent is visible. If there is no such place, then the inherited subprogram is not declared at all. An inherited subprogram that is not declared at all cannot be named in a call and cannot be overridden, but for a tagged type it is possible to dispatch to it.

7.6 User-Defined Assignment and Finalization

Replace paragraph 4: [8652/0020]

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

by:

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

Replace paragraph 11: [8652/0021]

For an extension_aggregate whose ancestor_part is a subtype_mark, Initialize is called on all controlled subcomponents of the ancestor part; if the type of the ancestor part is itself controlled, the Initialize procedure of the ancestor type is called, unless that Initialize procedure is abstract.

by:

For an extension_aggregate whose ancestor_part is a subtype_mark, for each controlled subcomponent of the ancestor part, either Initialize is called, or its initial value is assigned, as appropriate; if the type of the ancestor part is itself controlled, the Initialize procedure of the ancestor type is called, unless that Initialize procedure is abstract.

Insert after paragraph 17: [8652/0022]

For an assignment_statement, after the name and expression have been evaluated, and any conversion (including constraint checking) has been done, an anonymous object is created, and the value is assigned into it; that is, the assignment operation is applied. (Assignment includes value adjustment.) The target of the assignment_statement is then finalized. The value of the anonymous object is then assigned into the target of the assignment_statement. Finally, the anonymous object is finalized. As explained below, the implementation may eliminate the intermediate anonymous object, so this description subsumes the one given in 5.2, “Assignment Statements”.

the new paragraph:

Implementation Requirements

For an aggregate of a controlled type whose value is assigned, other than by an assignment_statement or a return_statement, the implementation shall not create a separate anonymous object for the aggregate. The aggregate value shall be constructed directly in the target of the assignment operation and Adjust is not called on the target object.

7.6.1 Completion and Finalization

Replace paragraph 13: [8652/0021; 8652/0023]

The anonymous objects created by function calls and by aggregates are finalized no later than the end of the innermost enclosing declarative_item or statement; if that is a compound_statement, they are finalized before starting the execution of any statement within the compound_statement.

by:

If the object_name in an object_renaming_declaration, or the actual parameter for a generic formal in out parameter in a generic_instantiation, denotes any part of an anonymous object created by a function call, the anonymous object is not finalized until after it is no longer accessible via any name. Otherwise, an anonymous object created by a function call or by an aggregate is finalized no later than the end of the innermost enclosing declarative_item or statement; if that is a compound_statement, the object is finalized before starting the execution of any statement within the compound_statement.

If a transfer of control or raising of an exception occurs prior to performing a finalization of an anonymous object, the anonymous object is finalized as part of the finalizations due to be performed for the object's innermost enclosing master.

Replace paragraph 14: [8652/0023]

It is a bounded error for a call on Finalize or Adjust to propagate an exception. The possible consequences depend on what action invoked the Finalize or Adjust operation:

by:

It is a bounded error for a call on Finalize or Adjust that occurs as part of object finalization or assignment to propagate an exception. The possible consequences depend on what action invoked the Finalize or Adjust operation:

Replace paragraph 16: [8652/0024]

- For an Adjust invoked as part of an assignment operation, any other adjustments due to be performed are performed, and then Program_Error is raised.

by:

- For an Adjust invoked as part of the initialization of a controlled object, other adjustments due to be performed might or might not be performed, and then Program_Error is raised. During its propagation, finalization might or might not be applied to objects whose Adjust failed. For an Adjust invoked as part of an assignment statement, any other adjustments due to be performed are performed, and then Program_Error is raised.

Insert after paragraph 17: [8652/0023]

- For a Finalize invoked as part of a call on an instance of Unchecked_Deallocation, any other finalizations due to be performed are performed, and then Program_Error is raised.

the new paragraphs:

- For a Finalize invoked as part of the finalization of the anonymous object created by a function call or **aggregate**, any other finalizations due to be performed are performed, and then Program_Error is raised.
- For a Finalize invoked due to reaching the end of the execution of a master, any other finalizations associated with the master are performed, and Program_Error is raised immediately after leaving the master.

Section 8: Visibility Rules

8.3 Visibility

Replace paragraph 9: [8652/0025]

Two homographs are not generally allowed immediately within the same declarative region unless one *overrides* the other (see Legality Rules below). A declaration overrides another homograph that occurs immediately within the same declarative region in the following cases:

by:

Two homographs are not generally allowed immediately within the same declarative region unless one *overrides* the other (see Legality Rules below). The only declarations that are *overridable* are the implicit declarations for predefined operators and inherited primitive subprograms. A declaration overrides another homograph that occurs immediately within the same declarative region in the following cases:

Replace paragraph 10: [8652/0025]

- An explicit declaration overrides an implicit declaration of a primitive subprogram, regardless of which declaration occurs first;

by:

- A declaration that is not overridable overrides one that is overridable, regardless of which declaration occurs first;

Replace paragraph 26: [8652/0025; 8652/0026]

An explicit declaration is illegal if there is a homograph occurring immediately within the same declarative region that is visible at the place of the declaration, and is not hidden from all visibility by the explicit declaration. Similarly, the `context_clause` for a subunit is illegal if it mentions (in a `with_clause`) some library unit, and there is a homograph of the library unit that is visible at the place of the corresponding stub, and the homograph and the mentioned library unit are both declared immediately within the same declarative region. These rules also apply to dispatching operations declared in the visible part of an instance of a generic unit. However, they do not apply to other overloadable declarations in an instance; such declarations may have type conformant profiles in the instance, so long as the corresponding declarations in the generic were not type conformant.

by:

A non-overridable declaration is illegal if there is a homograph occurring immediately within the same declarative region that is visible at the place of the declaration, and is not hidden from all visibility by the non-overridable declaration. In addition, a type extension is illegal if somewhere within its immediate scope it has two visible components with the same name. Similarly, the `context_clause` for a subunit is illegal if it mentions (in a `with_clause`) some library unit, and there is a homograph of the library unit that is visible at the place of the corresponding stub, and the homograph and the mentioned library unit are both declared immediately within the same declarative region. These rules also apply to dispatching operations declared in the visible part of an instance of a generic unit. However, they do not apply to other overloadable declarations in an instance; such declarations may have type conformant profiles in the instance, so long as the corresponding declarations in the generic were not type conformant.

8.5.1 Object Renaming Declarations

Replace paragraph 5: [8652/0017]

The renamed entity shall not be a subcomponent that depends on discriminants of a variable whose nominal subtype is unconstrained, unless this subtype is indefinite, or the variable is aliased. A slice of an array shall not be renamed if this restriction disallows renaming of the array.