# Information technology —Programming languages — Ada

## AMENDMENT 1 (Draft 1)

*Technologies de l'information —Langages de programmation — Ada*

*AMENDEMENT 1*

Amendment 1 to International Standard ISO/IEC 8652:1995 was prepared by AXE Consultants.

# Introduction

International Standard ISO/IEC 8652:1995 defines the Ada programming language.

This amendment modifies Ada by making changes and additions that improve:

- The safety of applications written in Ada;
- The portability of applications written in Ada;
- Interoperability with other languages and systems; and
- Accessibility and ease of transition from idioms in other programming and modeling languages.

This amendment incorporates the following major additions to the International Standard:

- Control of overriding to eliminate errors (see clause 8.3);
- A mechanism for writing C unions to make interfaces with C system easier (see clause x.x); and
- Type stubs to allow mutually dependent types (see clause x.x).

This Amendment is organized by sections corresponding to those in the International Standard. These sections include wording changes and additions to the International Standard. Clause and subclause headings are given for each clause that contains a wording change. Clauses and subclauses that do not contain any change or addition are omitted.

For each change, an *anchor* paragraph from the International Standard (as corrected by Technical Corrigendum 1) 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 International Standard - this font for text, this font for syntax, `and this font for Ada source code.`

**Disclaimer:**

**This document is a draft of a possible amendment to Ada 95 (International Standard ISO/IEC 8652:1995). This draft contains only proposals substantially approved by the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group (ARG). Many other important proposals are under consideration by the ARG. Neither the ARG nor any other group has determined which, if any, of these proposals will be included in the amendment. Any proposal may be substantially changed or withdrawn before this document begins standardization, and other proposals may be added. This document is not an official publication or work product of the ARG.**

# Section 1: General

No changes in this section.

# Section 2: Lexical Elements

No changes in this section.

# Section 3: Declarations and Types

## 3.10 Access Types

**Replace paragraph 9: [AI95-00225-01]**

A view of an object is defined to be *aliased* if it is defined by an object_declaration or component_definition with the reserved word **aliased**, or by a renaming of an aliased view. In addition, the dereference of an access-to-object value denotes an aliased view, as does a view conversion (see 4.6) of an aliased view. Finally, the current instance of a limited type, and a formal parameter or generic formal object of a tagged type are defined to be aliased. Aliased views are the ones that can be designated by an access value. If the view defined by an object_declaration is aliased, and the type of the object has discriminants, then the object is constrained; if its nominal subtype is unconstrained, then the object is constrained by its initial value. Similarly, if the object created by an allocator has discriminants, the object is constrained, either by the designated subtype, or by its initial value.

**by:**

A view of an object is defined to be *aliased* if it is defined by an object_declaration or component_definition with the reserved word **aliased**, or by a renaming of an aliased view. In addition, the dereference of an access-to-object value denotes an aliased view, as does a view conversion (see 4.6) of an aliased view. A current instance of a limited tagged type, a protected type, a task type, or a type that has the reserved word **limited** in its full definition is also defined to be aliased. Finally, a formal parameter or generic formal object of a tagged type is defined to be aliased. Aliased views are the ones that can be designated by an access value. If the view defined by an object_declaration is aliased, and the type of the object has discriminants, then the object is constrained; if its nominal subtype is unconstrained, then the object is constrained by its initial value. Similarly, if the object created by an allocator has discriminants, the object is constrained, either by the designated subtype, or by its initial value.

## 3.10.2 Operations of Access Types

**Replace paragraph 2: [AI95-00235-01]**

For an attribute_reference with attribute_designator Access (or Unchecked_Access -- see 13.10), the expected type shall be a single access type; the prefix of such an attribute_reference is never interpreted as an implicit_dereference. If the expected type is an access-to-subprogram type, then the expected profile of the prefix is the designated profile of the access type.

**by:**

For an attribute_reference with attribute_designator Access (or Unchecked_Access -- see 13.10), the expected type shall be a single access type A such that:

- A is an access-to-object type with designated type D and the type of the prefix is D'Class or is covered by D, or
- A is an access-to-subprogram type whose designated profile is type conformant with that of the prefix.

The prefix of such an attribute_reference is never interpreted as an implicit_dereference or parameterless function_call (see 4.1.4). The designated type or profile of the expected type of the attribute_reference is the expected type or profile for the prefix.

**Replace paragraph 32: [AI95-00229-01]**

P'Access yields an access value that designates the subprogram denoted by P. The type of P'Access is an access-to-subprogram type (*S*), as determined by the expected type. The accessibility level of P shall not be statically deeper than that of *S*. In addition to the places where Legality Rules normally

apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. The profile of P shall be subtype-conformant with the designated profile of *S*, and shall not be Intrinsic. If the subprogram denoted by P is declared within a generic body, *S* shall be declared within the generic body.

**by:**

P'Access yields an access value that designates the subprogram denoted by P. The type of P'Access is an access-to-subprogram type (*S*), as determined by the expected type. The accessibility level of P shall not be statically deeper than that of *S*. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. The profile of P shall be subtype-conformant with the designated profile of *S*, and shall not be Intrinsic. If the subprogram denoted by P is declared within a generic unit, and the expression P'Access occurs within the body of that generic unit or within the body of a generic unit declared within the declarative region of the generic, then the ultimate ancestor of *S* shall be a non-formal type declared within the generic unit.

# Section 4: Names and Expressions

## 4.6 Type Conversions

**Replace paragraph 9: [AI95-00246-01]**

If the target type is an array type, then the operand type shall be an array type. Further:

**by:**

If the target type is an array type, then the operand type shall be an array type. The target type and operation type shall have a common ancestor, or:

**Replace paragraph 12: [AI95-00246-01]**

- The component subtypes shall statically match; and

**by:**

- The component subtypes shall statically match;

**Replace paragraph 12.1: [AI95-00246-01]**

- In a view conversion, the target type and the operand type shall both or neither have aliased components.

**by:**

- Neither the target type nor the operand type shall be limited; and
- In a view conversion: the target type and the operand type shall both or neither have aliased components; and the operand type shall not have a tagged, private, or volatile subcomponent.

## 4.9 Static Expressions and Static Subtypes

**Replace paragraph 38: [AI95-00268-01]**

For a real static expression that is not part of a larger static expression, and whose expected type is not a descendant of a formal scalar type, the implementation shall round or truncate the value (according to the Machine_Rounds attribute of the expected type) to the nearest machine number of the expected type; if the value is exactly half-way between two machine numbers, any rounding shall be performed away from zero. If the expected type is a descendant of a formal scalar type, no special rounding or truncating is required - normal accuracy rules apply (see Annex G).

**by:**

For a real static expression that is not part of a larger static expression, and whose expected type is not a descendant of a formal scalar type, the implementation shall round or truncate the value (according to the Machine_Rounds attribute of the expected type) to the nearest machine number of the expected type; if the value is exactly half-way between two machine numbers, the rounding performed is implementation-defined. If the expected type is a descendant of a formal scalar type, no special rounding or truncating is required - normal accuracy rules apply (see Annex G).

*Implementation Advice*

For a real static expression that is not part of a larger static expression, and whose expected type is not a descendant of a formal scalar type, the rounding should be the same as the default rounding for the target system.

# Section 5: Statements

No changes in this section.

# Section 6: Subprograms

No changes in this section.

# Section 7: Packages

## 7.6 User-Defined Assignment and Finalization

**Replace paragraph 5: [AI95-00161-01]**

```
type Controlled is abstract tagged private;
```

**by:**

```
type Controlled is abstract tagged private;
pragma Preelaborable_Initialization(Controlled);
```

**Replace paragraph 7: [AI95-00161-01]**

```
type Limited_Controlled is abstract tagged limited private;
```

**by:**

```
type Limited_Controlled is abstract tagged limited private;
pragma Preelaborable_Initialization(Limited_Controlled);
```

# Section 8: Visibility Rules

## 8.3 Visibility

**Insert after paragraph 26: [AI95-00218-01]**

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.

**the new paragraphs:**

*Syntax*

The form of a pragma Explicit_Overriding is as follows:

**pragma** Explicit_Overriding;

The form of a pragma Overriding is as follows:

**pragma** Overriding [(designator)];

The form of a pragma Optional_Overriding is as follows:

**pragma** Optional_Overriding [(designator)];

Pragma Explicit_Overriding is a configuration pragma.

*Legality Rules*

Pragmas Overriding and Optional_Overriding shall immediately follow (except for other pragmas) the explicit declaration of a primitive operation. The optional designator of a pragma Overriding or Optional_Overriding shall be the same as the designator of the operation which it follows. Only one of the pragmas Overriding and Optional_Overriding shall be given for a single primitive operation.

A primitive operation to which pragma Overriding applies shall override another operation. In addition to the places where Legality Rules normally apply, this rule also applies in the private part of an instance of a generic unit.

The configuration pragma Explicit_Overriding applies to all declarations within compilation units to which it applies, except that in an instance of a generic unit, Explicit_Overriding applies if and only if it applies to the generic unit. At a place where a pragma Explicit_Overriding applies, an explicit subprogram_declaration to which neither pragma Overriding nor Optional_Overriding applies shall not be an overriding declaration. In addition to the places where Legality Rules normally apply, this rule also applies in the private part of an instance of a generic unit.

# Section 9: Tasks and Synchronization

## 9.6 Delay Statements, Duration, and Time

**Replace paragraph 10: [AI95-00161-01]**

```
package Ada.Calendar is
    type Time is private;
```

**by:**

```
package Ada.Calendar is
    type Time is private;
    pragma Preelaborable_Initialization(Time);
```

# Section 10: Program Structure and Compilation Issues

## 10.2.1 Elaboration Control

**Insert after paragraph 4: [AI95-00161-01]**

A pragma Preelaborate is a library unit pragma.

**the new paragraphs:**

The form of pragma Preelaborable_Initialization is as follows:

**pragma** Preelaborable_Initialization (direct_name);

**Replace paragraph 9: [AI95-00161-01]**

- The creation of a default-initialized object (including a component) of a descendant of a private type, private extension, controlled type, task type, or protected type with entry_declarations; similarly the evaluation of an extension_aggregate with an ancestor subtype_mark denoting a subtype of such a type.

**by:**

- The creation of an object (including a component) of a type which does not have preelaborable initialization. Similarly the evaluation of an extension_aggregate with an ancestor subtype_mark denoting a subtype of such a type.

**Insert after paragraph 11: [AI95-00161-01]**

If a pragma Preelaborate (or pragma Pure -- see below) applies to a library unit, then it is *preelaborated*. If a library unit is preelaborated, then its declaration, if any, and body, if any, are elaborated prior to all non-preelaborated library_items of the partition. The declaration and body of a preelaborated library unit, and all subunits that are elaborated as part of elaborating the library unit, shall be preelaborable. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. In addition, all compilation units of a preelaborated library unit shall depend semantically only on compilation units of other preelaborated library units.

**the new paragraphs:**

The following rules specify which entities have preelaborable initialization:

- The partial view of a private type or private extension, a protected type without entry_declarations, a generic formal private type, or a generic formal derived type, have preelaborable initialization if and only if the pragma Preelaborable_Initialization has been applied to them.
- A component (including a discriminant) of a record or protected type has preelaborable initialization if its declaration includes a default_expression whose execution does not perform any actions prohibited in preelaborable constructs as described above, or if its declaration does not include a default expression and its type has preelaborable initialization.
- A derived type has preelaborable initialization if its parent type has preelaborable initialization and (in the case of a derived record or protected type) if the non-inherited components all have preelaborable initialization. Moreover, a user-defined controlled type with an overridding Initialize procedure does not have preelaborable initialization.
- A view of a type has preelaborable initialization if it is an elementary type, an array type whose component type has preelaborable initialization, or a record type whose components all have preelaborable initialization.

A pragma Preelaborable_Initialization specifies that a type has preelaborable initialization. This pragma shall appear in the visible part of a package or generic package.

If the pragma appears in the first list of declarative_items of a package_specification, then the direct_name shall denote the first subtype of a private type, private extension, or protected type without entry_declarations, and the type shall be declared within the same package as the pragma. If the pragma is applied to a private type or a private extension, the full view of the type shall have preelaborable initialization. If the pragma is applied to a protected type, each component of the protected type shall have preelaborable initialization. In addition to the places where Legality Rules normally apply, these rules apply also in the private part of an instance of a generic unit.

If the pragma appears in a generic_formal_part, then the direct_name shall denote a generic formal private type or a generic formal derived type declared in the same generic_formal_part as the pragma. In a generic_instantiation the corresponding actual type shall have preelaborable initialization.

# Section 11: Exceptions

## 11.4.1 The Package Exceptions

**Replace paragraph 14: [AI95-00241-01]**

Raise_Exception and Reraise_Occurrence have no effect in the case of Null_Id or Null_Occurrence. Exception_Message, Exception_Identity, Exception_Name, and Exception_Information raise Constraint_Error for a Null_Id or Null_Occurrence.

**by:**

Raise_Exception and Reraise_Occurrence have no effect in the case of Null_Id or Null_Occurrence. Exception_Name raises Constraint_Error for a Null_Id. Exception_Message, Exception_Name, and Exception_Information raise Constraint_Error for a Null_Occurrence. Exception_Identity applied to Null_Occurrence returns Null_Id.

# Section 12: Generic Units

## 12.5 Formal Types

**Replace paragraph 8: [AI95-00233-01]**

The formal type also belongs to each class that contains the determined class. The primitive subprograms of the type are as for any type in the determined class. For a formal type other than a formal derived type, these are the predefined operators of the type. For an elementary formal type, the predefined operators are implicitly declared immediately after the declaration of the formal type. For a composite formal type, the predefined operators are implicitly declared either immediately after the declaration of the formal type, or later in its immediate scope according to the rules of 7.3.1. In an instance, the copy of such an implicit declaration declares a view of the predefined operator of the actual type, even if this operator has been overridden for the actual type. The rules specific to formal derived types are given in 12.5.1.

**by:**

The formal type also belongs to each class that contains the determined class. The primitive subprograms of the type are as for any type in the determined class. For a formal type other than a formal derived type, these are the predefined operators of the type. For an elementary formal type, the predefined operators are implicitly declared immediately after the declaration of the formal type. For a composite formal type, the predefined operators are implicitly declared either immediately after the declaration of the formal type, or later immediately within the declarative region in which the type is declared according to the rules of 7.3.1. In an instance, the copy of such an implicit declaration declares a view of the predefined operator of the actual type, even if this operator has been overridden for the actual type. The rules specific to formal derived types are given in 12.5.1.

## 12.5.1 Formal Private and Derived Types

**Replace paragraph 20: [AI95-00233-01]**

If the ancestor type is a composite type that is not an array type, the formal type inherits components from the ancestor type (including discriminants if a new discriminant_part is not specified), as for a derived type defined by a derived_type_definition (see 3.4).

**by:**

If the ancestor type is a composite type that is not an array type, the formal type inherits components from the ancestor type (including discriminants if a new discriminant_part is not specified), as for a derived type defined by a derived_type_definition (see 3.4 and 7.3.1).

**Replace paragraph 21: [AI95-00233-01]**

For a formal derived type, the predefined operators and inherited user-defined subprograms are determined by the ancestor type, and are implicitly declared at the earliest place, if any, within the immediate scope of the formal type, where the corresponding primitive subprogram of the ancestor is visible (see 7.3.1). In an instance, the copy of such an implicit declaration declares a view of the corresponding primitive subprogram of the ancestor of the formal derived type, even if this primitive has been overridden for the actual type. When the ancestor of the formal derived type is itself a formal type, the copy of the implicit declaration declares a view of the corresponding copied operation of the ancestor. In the case of a formal private extension, however, the tag of the formal type is that of the actual type, so if the tag in a call is statically determined to be that of the formal type, the body executed will be that corresponding to the actual type.

**by:**

For a formal derived type, the predefined operators and inherited user-defined subprograms are determined by the ancestor type, and are implicitly declared at the earliest place, if any, immediately within the declarative region in which the formal type is declared, where the corresponding primitive subprogram of the ancestor is visible (see 7.3.1). In an instance, the copy of such an implicit declaration declares a view of the corresponding

primitive subprogram of the ancestor of the formal derived type, even if this primitive has been overridden for the actual type. When the ancestor of the formal derived type is itself a formal type, the copy of the implicit declaration declares a view of the corresponding copied operation of the ancestor. In the case of a formal private extension, however, the tag of the formal type is that of the actual type, so if the tag in a call is statically determined to be that of the formal type, the body executed will be that corresponding to the actual type.

# Section 13: Representation Issues

## 13.3 Representation Attributes

**Delete paragraph 26: [AI95-00247-01]**

If an Alignment is specified for a composite subtype or object, this Alignment shall be equal to the least common multiple of any specified Alignments of the subcomponent subtypes, or an integer multiple thereof.

## 13.7 The Package System

**Replace paragraph 12: [AI95-00161-01]**

```
type Address is implementation-defined;
Null_Address : constant Address;
```

**by:**

```
type Address is implementation-defined;
pragma Preelaborable_Initialization(Address);
Null_Address : constant Address;
```

**In paragraph 15 replace: [AI95-00221-01]**

```
Default_Bit_Order : constant Bit_Order;
```

**by:**

```
Default_Bit_Order : constant Bit_Order := implementation-defined;
```

**Replace paragraph 35: [AI95-00221-01]**

See 13.5.3 for an explanation of Bit_Order and Default_Bit_Order.

**by:**

See 13.5.3 for an explanation of Bit_Order and Default_Bit_Order. Default_Bit_Order shall be a static constant.

## 13.11 Storage Management

**Replace paragraph 6: [AI95-00161-01]**

```
type Root_Storage_Pool is
    abstract new Ada.Controlled.Limited_Controlled with private;
```

**by:**

```
type Root_Storage_Pool is
    abstract new Ada.Controlled.Limited_Controlled with private;
pragma Preelaborable_Initialization(Root_Storage_Pool);
```

## 13.12 Pragma Restrictions

**Insert after paragraph 7: [AI95-00257-01]**

The set of restrictions is implementation defined.

**the new paragraphs:**

The following *restriction*_identifiers are language-defined (additional restrictions are defined in the Specialized Needs Annexes):

No_Implementation_Attributes
: There are no implementation-defined attributes. This restriction applies only to the current compilation or environment, not the entire partition.

No_Implementation_Pragmas
: There are no implementation-defined pragmas or pragma arguments. This restriction applies only to the current compilation or environment, not the entire partition.

## 13.13.1 The Package Streams

**Replace paragraph 3: [AI95-00161-01]**

```
type Root_Stream_Type is abstract tagged limited private;
```

**by:**

```
type Root_Stream_Type is abstract tagged limited private;
pragma Preelaborable_Initialization(Root_Stream_Type);
```

**Replace paragraph 8: [AI95-00227-01]**

The Read operation transfers Item'Length stream elements from the specified stream to fill the array Item. The index of the last stream element transferred is returned in Last. Last is less than Item'Last only if the end of the stream is reached.

**by:**

The Read operation transfers stream elements from the specified stream to fill the array Item. Elements are transferred until Item'Length elements have been transferred, or until the end of the stream is reached. If any elements are transferred, the index of the last stream element transferred is returned in Last. Otherwise, Item'First - 1 is returned in Last. Last is less than Item'Last only if the end of the stream is reached.

**Insert after paragraph 10: [AI95-00227-01]**

See A.12.1, ``The Package Streams.Stream_IO'' for an example of extending type Root_Stream_Type.

**the new paragraph:**

If the end of stream has been reached, and Item'First is Stream_Element_Offset'First, Read will raise Constraint_Error.

## 13.13.2 Stream-Oriented Attributes

**Insert after paragraph 28: [AI95-00260-01]**

For every subtype S'Class of a class-wide type *T*'Class:

**the new paragraphs:**

S'Class'Tag_Write

S'Class'Tag_Write denotes a procedure with the following specification:

```
procedure S'Class'Tag_Write (
    Stream : access Streams.Root_Stream_Type'Class;
    Tag : Ada.Tags.Tag);
```

S'Class'Tag_Write writes the value of Tag to Stream.

S'Class'Tag_Read

S'Class'Tag_Read denotes a function with the following specification:

```
function S'Class'Tag_Write (
     Stream : access Streams.Root_Stream_Type'Class)
     return Ada.Tags.Tag;
```

S'Class'Tag_Read reads a tag from Stream, and returns its value.

The default implementations of the Tag_Write and Tag_Read operate as follows:

- If *T* is a derived type with parent type *P*, the default implementation of Tag_Write calls *P*'Class'Tag_Write, and the default implementation of Tag_Read calls *P*'Class'Tag_Read;
- Otherwise, the default implementation of Tag_Write calls String'Output(*Stream*, Tags.External_Tag(*Tag*)) -- see 3.9. The default implementation of Tag_Read returns the value of Tags.Internal_Tag(String'Input(*Stream*)).

**Replace paragraph 31: [AI95-00260-01]**

First writes the external tag of *Item* to *Stream* (by calling String'Output(Tags.External_Tag(*Item*'Tag) -- see 3.9) and then dispatches to the subprogram denoted by the Output attribute of the specific type identified by the tag.

**by:**

First writes the external tag of *Item* to *Stream* (by calling S'Tag_Write(*Stream*, *Item*'Tag)) and then dispatches to the subprogram denoted by the Output attribute of the specific type identified by the tag.

**Replace paragraph 34: [AI95-00260-01]**

First reads the external tag from *Stream* and determines the corresponding internal tag (by calling Tags.Internal_Tag(String'Input(*Stream*)) -- see 3.9) and then dispatches to the subprogram denoted by the Input attribute of the specific type identified by the internal tag; returns that result.

**by:**

First reads the external tag from *Stream* and determines the corresponding internal tag (by calling S'Tag_Read(*Stream*)) and then dispatches to the subprogram denoted by the Input attribute of the specific type identified by the internal tag; converts that result to S'Class and returns it.

**Insert after paragraph 38: [AI95-00260-01]**

User-specified attributes of S'Class are not inherited by other class-wide types descended from S.

**the new paragraph:**

User-specified Tag_Read and Tag_Write attributes should raise an exception if presented with a tag value not in S'Class.

# Annex A: Predefined Language Environment

## A.4.2 The Package Strings.Maps

**Replace paragraph 4: [AI95-00161-01]**

```
-- Representation for a set of Wide_Character values:
type Wide_Character_Set is private;
```

**by:**

```
-- Representation for a set of Wide_Character values:
type Wide_Character_Set is private;
pragma Preelaborable_Initialization(Wide_Character_Set);
```

**Replace paragraph 4: [AI95-00161-01]**

```
-- Representation for a set of character values:
type Character_Set is private;
```

**by:**

```
-- Representation for a set of character values:
type Character_Set is private;
pragma Preelaborable_Initialization(Character_Set);
```

**Replace paragraph 20: [AI95-00161-01]**

```
-- Representation for a Wide_Character to Wide_Character mapping:
type Wide_Character_Mapping is private;
```

**by:**

```
-- Representation for a Wide_Character to Wide_Character mapping:
type Wide_Character_Mapping is private;
pragma Preelaborable_Initialization(Wide_Character_Mapping);
```

**Replace paragraph 20: [AI95-00161-01]**

```
-- Representation for a character to character mapping:
type Character_Mapping is private;
```

**by:**

```
-- Representation for a character to character mapping:
type Character_Mapping is private;
pragma Preelaborable_Initialization(Character_Mapping);
```

## A.4.4 Bounded-Length String Handling

**Replace paragraph 101: [AI95-00238-01]**

Returns the slice at positions Low through High in the string represented by Source; propagates Index_Error if Low > Length(Source)+1 or High > Length(Source).

**by:**

Returns the slice at positions Low through High in the string represented by Source; propagates Index_Error if Low > Length(Source)+1 or High > Length(Source). The bounds of the returned string are Low and High.

## A.4.5 Unbounded-Length String Handling

**Replace paragraph 4: [AI95-00161-01]**

```
type Unbounded_String is private;
```

**by:**

```
type Unbounded_String is private;
pragma Preelaborable_Initialization(Unbounded_String);
```

## A.5.3 Attributes of Floating Point Types

**Insert after paragraph 41: [AI95-00267-01]**

The function yields the integral value nearest to *X*, rounding toward the even integer if *X* lies exactly halfway between two integers. A zero result has the sign of *X* when S'Signed_Zeros is True.

**the new paragraphs:**

S'Machine_Rounding

S'Machine_Rounding denotes a function with the following specification:

```
function S'Machine_Rounding (X : T)
    return T
```

The function yields the integral value nearest to *X*. If *X* lies exactly halfway between two integers, one of those integers is returned, but which of them is returned is unspecified. A zero result has the sign of *X* when S'Signed_Zeros is True. This function provides access to the rounding behavior which is most efficient on the target processor.

## A.10.6 Get and Put Procedures

**In paragraph 5 replace: [AI95-00223-01]**

Input-output of enumeration values uses the syntax of the corresponding lexical elements. Any Get procedure for an enumeration type begins by skipping any leading blanks, or line or page terminators. Get procedures for numeric or enumeration types start by skipping leading blanks, where a *blank* is defined as a space or a horizontal tabulation character. Next, characters are input only so long as the sequence input is an initial sequence of an identifier or of a character literal (in particular, input ceases when a line terminator is encountered). The character or line terminator that causes input to cease remains available for subsequent input.

**by:**

Input-output of enumeration values uses the syntax of the corresponding lexical elements. Any Get procedure for an enumeration type begins by skipping any leading blanks, or line or page terminators. A *blank* is defined as a space or a horizontal tabulation character. Next, characters are input only so long as the sequence input is an initial sequence of an identifier or of a character literal (in particular, input ceases when a line terminator is encountered). The character or line terminator that causes input to cease remains available for subsequent input.

## A.12.1 The Package Streams.Stream_IO

**Replace paragraph 28.1: [AI95-00085-01]**

The Set_Mode procedure changes the mode of the file. If the new mode is Append_File, the file is positioned to its end; otherwise, the position in the file is unchanged.

**by:**

The Set_Mode procedure sets the mode of the file. If the new mode is Append_File, the file is positioned to its end; otherwise, the position in the file is unchanged.

# Annex B: Interface to Other Languages

## B.3 Interfacing with C

**Replace paragraph 50: [AI95-00258-01]**

The result of To_C is a char_array value of length Item'Length (if Append_Nul is False) or Item'Length+1 (if Append_Nul is True). The lower bound is 0. For each component Item(I), the corresponding component in the result is To_C applied to Item(I). The value nul is appended if Append_Nul is True.

**by:**

The result of To_C is a char_array value of length Item'Length (if Append_Nul is False) or Item'Length+1 (if Append_Nul is True). The lower bound is 0. For each component Item(I), the corresponding component in the result is To_C applied to Item(I). The value nul is appended if Append_Nul is True. If Append_Nul is False and Item'Length is 0, then To_C propagates Constraint_Error.

### B.3.1 The Package Interfaces.C.Strings

**Replace paragraph 5: [AI95-00161-01]**

```
type Chars_Ptr is private;
```

**by:**

```
type Chars_Ptr is private;
pragma Preelaborable_Initialization(Chars_Ptr);
```

**Replace paragraph 50: [AI95-00242-01]**

Equivalent to Update(Item, Offset, To_C(Str), Check).

**by:**

Equivalent to Update(Item, Offset, To_C(Str, Append_Nul => False), Check).

# Annex C: Systems Programming

## C.3.1 Protected Procedure Handlers

**Replace paragraph 8: [AI95-00253-01]**

The Interrupt_Handler pragma is only allowed immediately within a protected_definition. The corresponding protected_type_declaration shall be a library level declaration. In addition, any object_declaration of such a type shall be a library level declaration.

**by:**

The Interrupt_Handler pragma is only allowed immediately within a protected_definition where the corresponding subprogram is declared. The corresponding protected_type_declaration or single_protected_declaration shall be a library level declaration. In addition, any object_declaration of such a type shall be a library level declaration.

## C.4 Preelaboration Requirements

**Insert after paragraph 4: [AI95-00161-01]**

- Any subtype_mark denotes a statically constrained subtype, with statically constrained subcomponents, if any;

**the new paragraph:**

- No subtype_mark denotes a controlled type, a private type, a private extension, a generic formal private type, a generic formal derived type, or a descendant of such a type;

## C.6 Shared Variable Control

**Replace paragraph 7: [AI95-00272-01]**

An *atomic* type is one to which a pragma Atomic applies. An *atomic* object (including a component) is one to which a pragma Atomic applies, or a component of an array to which a pragma Atomic_Components applies, or any object of an atomic type.

**by:**

An *atomic* type is one to which a pragma Atomic applies. An *atomic* object (including a component) is one to which a pragma Atomic applies, or a component of an array to which a pragma Atomic_Components applies, or any object of an atomic type, other than objects obtained by evaluating a slice.

# Annex D: Real-Time Systems

No changes in this section.

# Annex E: Distributed Systems

## E.2.2 Remote Types Library Units

**Replace paragraph 8: [AI95-00240-01]**

- if the full view of a type declared in the visible part of the library unit has a part that is of a non-remote access type, then that access type, or the type of some part that includes the access type subcomponent, shall have user-specified Read and Write attributes.

**by:**

- if the full view of a type declared in the visible part of the library unit has a part that is of a non-remote access type, then that access type, or the type of some part that includes the access type subcomponent, shall have Read and Write attributes specified by a visible attribute_definition_clause.

**Replace paragraph 14: [AI95-00240-01]**

- The primitive subprograms of the corresponding specific limited private type shall only have access parameters if they are controlling formal parameters; each non-controlling formal parameter shall have either a nonlimited type or a type with Read and Write attributes specified via an attribute_definition_clause;

**by:**

- The primitive subprograms of the corresponding specific limited private type shall only have access parameters if they are controlling formal parameters; each non-controlling formal parameter shall have either a nonlimited type or a type with available Read and Write attributes (see 13.13.2);

## E.2.3 Remote Call Interface Library Units

**Replace paragraph 14: [AI95-00240-01]**

- it shall not be, nor shall its visible part contain, a subprogram (or access-to-subprogram) declaration whose profile has an access parameter, or a formal parameter of a limited type unless that limited type has user-specified Read and Write attributes;

**by:**

- it shall not be, nor shall its visible part contain, a subprogram (or access-to-subprogram) declaration whose profile has an access parameter, or a formal parameter of a limited type unless that limited type has available Read and Write attributes (see 13.13.2);

## E.5 Partition Communication Subsystem

**Replace paragraph 1: [AI95-00273-01]**

The *Partition Communication Subsystem* (PCS) provides facilities for supporting communication between the active partitions of a distributed program. The package System.RPC is a language-defined interface to the PCS. An implementation conforming to this Annex shall use the RPC interface to implement remote subprogram calls.

**by:**

The *Partition Communication Subsystem* (PCS) provides facilities for supporting communication between the active partitions of a distributed program. The package System.RPC is a language-defined interface to the PCS.

**Insert after paragraph 27: [AI95-00273-01]**

A body for the package System.RPC need not be supplied by the implementation.

**the new paragraph:**

An alternative declaration is allowed for package System.RPC as long as it provides a set of operations that is substantially equivalent to the specification defined in this clause.

# Annex F: Information Systems

No changes in this section.

# Annex G: Numerics

## G.1.1 Complex Types

**Replace paragraph 4: [AI95-00161-01]**

```
type Imaginary is private;
```

**by:**

```
type Imaginary is private;
pragma Preelaborable_Initialization(Imaginary);
```

## G.1.2 Complex Elementary Functions

**Replace paragraph 15: [AI95-00185-01]**

The real (resp., imaginary) component of the result of the Arcsin and Arccos (resp., Arctanh) functions is discontinuous as the parameter X crosses the real axis to the left of -1.0 or the right of 1.0.

**by:**

The imaginary component of the result of the Arcsin, Arccos, and Arctanh functions is discontinuous as the parameter X crosses the real axis to the left of -1.0 or the right of 1.0.

**Replace paragraph 16: [AI95-00185-01]**

The real (resp., imaginary) component of the result of the Arctan (resp., Arcsinh) function is discontinuous as the parameter X crosses the imaginary axis below -*i* or above *i*.

**by:**

The real component of the result of the Arctan and Arcsinh functions is discontinuous as the parameter X crosses the imaginary axis below -*i* or above *i*.

**Replace paragraph 17: [AI95-00185-01]**

The real component of the result of the Arccot function is discontinuous as the parameter X crosses the imaginary axis between -*i* and *i*.

**by:**

The real component of the result of the Arccot function is discontinuous as the parameter X crosses the imaginary axis below -*i* or above *i*.

**Replace paragraph 20: [AI95-00185-01]**

The computed results of the mathematically multivalued functions are rendered single-valued by the following conventions, which are meant to imply the principal branch:

**by:**

The computed results of the mathematically multivalued functions are rendered single-valued by the following conventions, which are meant to imply that the principal branch is an analytic continuation of the corresponding real-valued function in Ada.Numerics.Generic_Elementary_Functions. (For Arctan and Arccot, the single-argument function in question is that obtained from the two-argument version by fixing the second argument to be its default value.)

# Annex H: Safety and Security

No changes in this section.

# Annex J: Obsolescent Features

No changes in this section.