What’s New in Ada 2022
Maxim Reznik

LEARN.
ADACORE.COM
What's New in Ada 2022

Release 2022-11

Maxim Reznik

Nov 18, 2022
# CONTENTS:

1 Introduction 3
   1.1 References 3

2 'Image attribute for any type 5
   2.1 'Image attribute for a value 5
   2.2 'Image attribute for any type 5
   2.3 References 6

3 Redefining the 'Image attribute 7
   3.1 What's the Root_Buffer_Type? 8
   3.2 Outdated draft implementation 8
   3.3 References 8

4 User-Defined Literals 9
   4.1 Turn Ada into JavaScript 10
   4.2 References 11

5 Advanced Array Aggregates 13
   5.1 Square brackets 13
   5.2 Iterated Component Association 14
   5.3 References 15

6 Container Aggregates 17
   6.1 References 21

7 Delta Aggregates 23
   7.1 Delta aggregate for records 23
   7.2 Delta aggregate for arrays 23
   7.3 References 24

8 Target Name Symbol (@) 25
   8.1 Alternatives 27
   8.2 References 27

9 Enumeration representation 29
   9.1 Literal positions 29
   9.2 Representation values 30
   9.3 Before Ada 2022 31
   9.4 References 32

10 Big Numbers 33
    10.1 Big Integers 33
    10.2 Tiny RSA implementation 33
    10.3 Big Reals 35
    10.4 References 35
This course presents an overview of the new features of the latest Ada 2022 standard. This document was written by Maxim Reznik and reviewed by Richard Kenner.
INTRODUCTION

This is a collection of short code examples demonstrating new features of the Ada 2022 Standard\(^2\) as they are implemented in GNAT Ada compiler.

To use some of these features, you may need to use a compiler command line switch or pragma. Compilers starting with GNAT Community Edition 2021\(^3\) or GCC 11\(^4\) use `pragma Ada_2022;` or the `-gnat2022` switch. Older compilers use `pragma Ada_2020;` or `-gnat2020`. To use the square brackets syntax or `*Reduce` expressions, you need `pragma Extensions_Allowed (On);` or the `-gnatX` switch.

1.1 References

- Draft Ada 2022 Standard\(^5\)
- Ada 202x support in GNAT\(^6\) blog post

---

\(^2\) http://www.ada-auth.org/standards/22aarm/html/AA-TTL.html
\(^3\) https://blog.adacore.com/gnat-community-2021-is-here
\(^4\) https://gcc.gnu.org/gcc-11/
\(^6\) https://blog.adacore.com/ada-202x-support-in-gnat
Chapter Two

'IMAGE ATTRIBUTE FOR ANY TYPE

Note: Attribute 'Image for any type is supported by

- GNAT Community Edition 2020 and latter
- GCC 11

2.1 'Image attribute for a value

Since the publication of the Technical Corrigendum 1\(^7\) in February 2016, the 'Image attribute can now be applied to a value. So instead of My_Type'Image (Value), you can just write Value'Image, as long as the Value is a name\(^8\). These two statements are equivalent:

```ada
Ada.Text_IO.Put_Line (Ada.Text_IO.Page_Length'Image);
```

2.2 'Image attribute for any type

In Ada 2022, you can apply the 'Image attribute to any type, including records, arrays, access types, and private types. Let's see how this works. We'll define array, record, and access types and corresponding objects and then convert these objects to strings and print them:

Listing 1: main.adb

```ada
pragma Ada_2022;
with Ada.Text_IO;

procedure Main is
  type Vector is array (Positive range <>) of Integer;
  V1 : aliased Vector := [1, 2, 3];
  type Text_Position is record
    Line, Column : Positive;
  end record;
```

\(^7\) https://reznikmm.github.io/ada-auth/rm-4-NC/RM-0-1.html

\(^8\) https://reznikmm.github.io/ada-auth/rm-4-NC/RM-4-1.html#S0091
What's New in Ada 2022

(continued from previous page)

```ada
   Pos : constant Text_Position := (Line => 10, Column => 3);
   type Vector_Access is access all Vector;
   V1_Ptr : constant Vector_Access := V1'Access;
begin
   Ada.Text_IO.Put_Line (V1'Image);
   Ada.Text_IO.Put_Line (Pos'Image);
   Ada.Text_IO.New_Line;
   Ada.Text_IO.Put_Line (V1_Ptr'Image);
end Main;
```

 Runtime output

```
[ 1, 2, 3]
(LINE => 10,
 COLUMN => 3)
(access 7ffda508898)
```

```
$ gprbuild -q -P main.gpr
   Build completed successfully.
$ ./main
   [ 1, 2, 3]
   (LINE => 10,
    COLUMN => 3)
   (access 7fff64b23988)
```

Note the square brackets in the array image output. In Ada 2022, array aggregates could be written this way (page 13)!

### 2.3 References

- ARM 4.10 Image Attributes
- AI12-0020-1

---

10 [http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/ai12-0020-1.txt](http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/ai12-0020-1.txt)
REDEFINING THE 'IMAGE ATTRIBUTE

In Ada 2022, you can redefine 'Image attribute for your type, though the syntax to do this has been changed several times. Let's see how it works in GNAT Community 2021.

**Note:** Redefining attribute 'Image is supported by

- GNAT Community Edition 2021 (using Text_Buffers)
- GCC 11 (using Text_Output.Utils)

In our example, let's redefine the 'Image attribute for a location in source code. To do this, we provide a new Put_Image aspect for the type:

Listing 1: main.adb

```ada
pragma Ada_2022;
with Ada.Text_IO;
with Ada.Strings.Text_Buffers;

procedure Main is
  type Source_Location is record
    Line : Positive;
    Column : Positive;
  end record
  with Put_Image => My_Put_Image;

procedure My_Put_Image
  (Output : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
   Value : Source_Location);

procedure My_Put_Image
  (Output : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
   Value : Source_Location)
is
  Line : constant String := Value.Line'Image;
  Column : constant String := Value.Column'Image;
  Result : constant String :=
    Line (2 .. Line'Last) & ':' & Column (2 .. Column'Last);
begin
  Output.Put (Result);
end My_Put_Image;

Line_10 : constant Source_Location := (Line => 10, Column => 1);

begin
  (continues on next page)
```
What's New in Ada 2022

33 Ada.Text_IO.Put_Line (Line_10'Image);
end Main;

Runtime output

10:1

3.1 What's the Root_Buffer_Type?

Let's see how it's defined in the Ada.Strings.Text_Buffers package.

type Root_Buffer_Type is abstract tagged limited private;

procedure Put
  (Buffer : in out Root_Buffer_Type;
     Item  : in  String) is abstract;

In addition to Put, there are also Wide_Put, Wide_Wide_Put, Put_UTF_8, Wide_Put_UTF_16. And also New_Line, Increase_Indent, Decrease_Indent.

3.2 Outdated draft implementation

GNAT Community Edition 2020 and GCC 11 both provide a draft implementation that's incompatible with the Ada 2022 specification. For those versions, My_Put_Image looks like:

procedure My_Put_Image
  (Sink   : in out Ada.Strings.Text_Output.Sink'Class;
     Value : Source_Location)
is
  Line  : constant String := Value.Line'Image;
  Column: constant String := Value.Column'Image;
  Result: constant String :=
    Line (2 .. Line'Last) & ':' & Column (2 .. Column'Last);
begin
  Ada.Strings.Text_Output.Utils.Put_UTF_8 (Sink, Result);
end My_Put_Image;

3.3 References

- ARM 4.10 Image Attributes\(^{11}\)
- AI12-0020-1\(^{12}\)
- AI12-0384-2\(^{13}\)

---

\(^{11}\) http://www.ada-auth.org/standards/22aarm/html/AA-4-10.html
\(^{12}\) http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0020-1.TXT
\(^{13}\) http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/AI12-0384-2.TXT
CHAPTER
FOUR

USER-DEFINED LITERALS

Note: User-defined literals are supported by

• GNAT Community Edition 2020
• GCC 11

In Ada 2022, you can define string, integer, or real literals for your types. The compiler will convert such literals to your type at run time using a function you provide. To do so, specify one or more new aspects:

• Integer_Literal
• Real_Literal
• String_Literal

For our example, let's define all three for a simple type and see how they work. For simplicity, we use a Wide_Wide_String component for the internal representation:

Listing 1: main.adb

```ada
pragma Ada_2022;
with Ada.Wide_Wide_Text_IO;
with Ada.Characters.Conversions;

procedure Main is
  type My_Type (Length : Natural) is record
    Value : Wide_Wide_String (1 .. Length);
  end record
  with String_Literal => From_String,
  Real_Literal => From_Real,
  Integer_Literal => From_Integer;

  function From_String (Value : Wide_Wide_String) return My_Type is
    ((Length => Value'Length, Value => Value));
  function From_Real (Value : String) return My_Type is
    ((Length => Value'Length,
      Value => Ada.Characters.Conversions.To_Wide_Wide_String (Value)));
  function From_Integer (Value : String) return My_Type renames From_Real;

  procedure Print (Self : My_Type) is
    begin
      Ada.Wide_Wide_Text_IO.Put_Line (Self.Value);
  end Print;
```

(continues on next page)
What's New in Ada 2022

(continued from previous page)

begin
  Print ("Test " "string" ");
  Print (123);
  Print (16#DEAD_BEEF#);
  Print (2.99_792_458e+8);
end Main;

Runtime output

Test "string"
123
16#DEAD_BEEF#
2.99_792_458e+8

As you see, real and integer literals are converted to strings while preserving the formatting
in the source code, while string literals are decoded: From_String is passed the specified
string value. In all cases, the compiler translates these literals into function calls.

4.1 Turn Ada into JavaScript

Do you know that '5'+3 in JavaScript is 53?

> '5'+3
'53'

Now we can get the same result in Ada! But before we do, we need to define a custom +
operator:

Listing 2: main.adb

pragma Ada_2022;
with Ada.Wide_Wide_Text_IO;
with Ada.Characters.Conversions;

procedure Main is
  type My_Type (Length : Natural) is record
    Value : Wide_Wide_String (1 .. Length);
  end record
  with String_Literal => From_String,
         Real_Literal => From_Real,
         Integer_Literal => From_Integer;

  function "+" (Left, Right : My_Type) return My_Type is
    (Left.Length + Right.Length, Left.Value & Right.Value);

  function From_String (Value : Wide_Wide_String) return My_Type is
    ((Length => Value'Length, Value => Value));

  function From_Real (Value : String) return My_Type is
    ((Length => Value'Length,
      Value => Ada.Characters.Conversions.To_Wide_Wide_String (Value)));

  function From_Integer (Value : String) return My_Type renames From_Real;

  procedure Print (Self : My_Type) is
begin
  Ada.Wide_Wide_Text_IO.Put_Line (Self.Value);
end Print;

begin
  Print ("5" + 3);
end Main;

**Runtime output**

53

Jokes aside, this feature is very useful. For example it allows a "native-looking API" for *big integers* (page 33).

## 4.2 References

- ARM 4.2.1 User-Defined Literals\(^{14}\)
- AI12-0249-1\(^{15}\)
- AI12-0342-1\(^{16}\)


\(^{15}\) [http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0249-1.TXT](http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0249-1.TXT)

\(^{16}\) [http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0342-1.TXT](http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0342-1.TXT)
Note: These array aggregates are supported by
- GNAT Community Edition 2020
- GCC 11

### 5.1 Square brackets

In Ada 2022, you can use square brackets in array aggregates. Using square brackets simplifies writing both empty aggregates and single-element aggregates. Consider this:

Listing 1: show_square_brackets.ads

```ada
pragma Ada_2022;
pragma Extensions_Allowed (On);

package Show_Square_Brackets is
  type Integer_Array is array (Positive range <>) of Integer;
  Old_Style_Empty : Integer_Array := (1 .. 0 => <>);
  New_STYLE_Empty : Integer_Array := [ ];
  Old_Style_One_Item : Integer_Array := (1 => 5);
  New_STYLE_One_Item : Integer_Array := [5];
end Show_Square_Brackets;
```

Build output

```
show_square_brackets.ads:8:39: warning: array aggregate using () is an obsolescent syntax, use [] instead [-gnatwj]
show_square_brackets.ads:11:42: warning: array aggregate using () is an obsolescent syntax, use [] instead [-gnatwj]
```

**Short summary for parentheses and brackets**

- Record aggregates use parentheses
- **Container aggregates** (page 17) use square brackets
- Array aggregates can use both square brackets and parentheses, but parentheses usage is obsolescent
5.2 Iterated Component Association

There is a new kind of component association:

Vector : Integer_Array := [for J in 1 .. 5 => J * 2];

This association starts with for keyword, just like a quantified expression. It declares an index parameter that you can use in the computation of a component. Iterated component associations can nest and can be nested in another association (iterated or not). Here we use this to define a square matrix:

Matrix : array (1 .. 3, 1 .. 3) of Positive :=
[for J in 1 .. 3 =>
 [for K in 1 .. 3 => J * 10 + K]];  

Iterated component associations in this form provide both element indices and values, just like named component associations:

Data : Integer_Array (1 .. 5) :=
[for J in 2 .. 3 => J, 5 => 5, others => 0];

Here Data contains (0, 2, 3, 0, 5), not (2, 3, 5, 0, 0). Another form of iterated component association corresponds to a positional component association and provides just values, but no element indices:

Vector_2 : Integer_Array := [for X of Vector => X / 2];

You cannot mix these forms in a single aggregate. It’s interesting that such aggregates were originally proposed more than 25 years ago!

Complete code snippet:

Listing 2: show_iterated_component_association.adb

```ada
pragma Ada_2022;
pragma Extensions_Allowed (On); -- for square brackets
with Ada.Text_IO;

procedure Show_Iterated_Component_Association is

   type Integer_Array is array (Positive range <>) of Integer;

   Old_Style_Empty : Integer_Array := (1 .. 0 => <>);
   New_Style_Empty : Integer_Array := [ ];

   Old_Style_One_Item : Integer_Array := (1 => 5);
   New_Style_One_Item : Integer_Array := [5];

   Vector : constant Integer_Array := [for J in 1 .. 5 => J * 2];

   Matrix : constant array (1 .. 3, 1 .. 3) of Positive :=
     [for J in 1 .. 3 =>
      [for K in 1 .. 3 => J * 10 + K]];  

   Data : constant Integer_Array (1 .. 5) :=
     [for J in 2 .. 3 => J, 5 => 5, others => 0];

   Vector_2 : constant Integer_Array := [for X of Vector => X / 2];
```

(continues on next page)
begin
    Ada.Text_IO.Put_Line (Vector'Image);
    Ada.Text_IO.Put_Line (Matrix'Image);
    Ada.Text_IO.Put_Line (Data'Image);
    Ada.Text_IO.Put_Line (Vector_2'Image);
end Show_Iterated_Component_Association;

Build output

show_iterated_component_association.adb:10:04: warning: variable "Old_Style_Empty"
 is not referenced [-gnatwu]
show_iterated_component_association.adb:10:39: warning: array aggregate using ()
 is an obsolescent syntax, use [] instead [-gnatwj]
show_iterated_component_association.adb:11:04: warning: variable "New_Style_Empty"
 is not referenced [-gnatwu]
show_iterated_component_association.adb:13:04: warning: variable "Old_Style_One_Item"
 is not referenced [-gnatwu]
show_iterated_component_association.adb:13:42: warning: array aggregate using ()
 is an obsolescent syntax, use [] instead [-gnatwj]
show_iterated_component_association.adb:14:04: warning: variable "New_STYLE_ONE_ITEM"
 is not referenced [-gnatwu]

Runtime output

[ 2, 4, 6, 8, 10]
[ 11, 12, 13],
[ 21, 22, 23],
[ 31, 32, 33]]
[ 0, 2, 3, 0, 5]
[ 1, 2, 3, 4, 5]

5.3 References

- ARM 4.3.3 Array Aggregates
- AI12-0212-1
- AI12-0306-1

\[17 \text{http://www.ada-auth.org/standards/22aarm/html/AA-4-3-3.html}
\[18 \text{http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0212-1.TXT}
\[19 \text{http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0306-1.TXT}
CHAPTER SIX

CONTAINER AGGREGATES

Note: Container aggregates are supported by

- GNAT Community Edition 2021
- GCC 11

Ada 2022 introduces container aggregates, which can be used to easily create values for vectors, lists, maps, and other aggregates. For containers such as maps, the aggregate must use named associations to provide keys and values. For other containers it uses positional associations. Only square brackets are allowed. Here's an example:

Listing 1: main.adb

```ada
pragma Ada_2022;
with Ada.Text_IO;
with Ada.Containers.Vectors;
with Ada.Containers.Ordered_Maps;

procedure Main is
  package Int_Vectors is new Ada.Containers.Vectors (Positive, Integer);
  X : constant Int_Vectors.Vector := [1, 2, 3];

  package Float_Maps is new Ada.Containers.Ordered_Maps (Integer, Float);
  Y : constant Float_Maps.Map := [10 => 1.0, 0 => 2.5, 10 => 5.51];

begin
  Ada.Text_IO.Put_Line (X'Image);
  Ada.Text_IO.Put_Line (Y'Image);
end Main;
```

Runtime output

```
[ 1, 2, 3]
[-10 => 1.00000E+00, 0 => 2.50000E+00, 10 => 5.51000E+00]
```

At runtime, the compiler creates an empty container and populates it with elements one by one. If you define a new container type, you can specify a new Aggregate aspect to enable container aggregates for your container and let the compiler know what subprograms to use to construct the aggregate:
What's New in Ada 2022

Listing 2: main.adb

```ada
pragma Ada_2022;

procedure Main is

   package JSON is
      type JSON_VALUE is private
         with Integer_Literal => To_JSON_Value;

      function To_JSON_Value (Text : String) return JSON_VALUE;

      type JSON_Array is private
         with Aggregate => (Empty => New_JSON_Array,
                             AddUnnamed => Append);

      function New_JSON_Array return JSON_Array;

      procedure Append
         (Self : in out JSON_Array;
          Value : JSON_VALUE) is null;

      private
         type JSON_Value is null record;
         type JSON_Array is null record;

      function To_JSON_Value (Text : String) return JSON_VALUE
         is (null record);

      function New_JSON_Array return JSON_Array is (null record);
   end JSON;

   List : JSON.JSON_Array := [1, 2, 3];

begin
   -- Equivalent old initialization code
   List := JSON.New_JSON_Array;
   JSON.Append (List, 1);
   JSON.Append (List, 2);
   JSON.Append (List, 3);
end Main;
```

The equivalent for maps is:

Listing 3: main.adb

```ada
pragma Ada_2022;

procedure Main is

   package JSON is
      type JSON_VALUE is private
         with Integer_Literal => To_JSON_Value;

      function To_JSON_Value (Text : String) return JSON_VALUE;

      type JSON_Object is private
         with Aggregate => (Empty => New_JSON_Object,
                             AddNamed => Insert);

      function New_JSON_Object return JSON_Object;
   end JSON;

begin
```

(continues on next page)
procedure Insert
(Self : in out JSON_Object;
Key : Wide_Wide_String;
Value : JSON_Value) is null;

private

type JSON_Value is null record;
type JSON_Object is null record;

function To_JSON_Value (Text : String) return JSON_Value
is (null record);

function New_JSON_Object return JSON_Object is (null record);
end JSON;

Object : JSON.JSON_Object := [
"a" => 1, "b" => 2, "c" => 3];

begin
-- Equivalent old initialization code
Object := JSON.New_JSON_Object;
JSON.Insert (Object, "a", 1);
JSON.Insert (Object, "b", 2);
JSON.Insert (Object, "c", 3);
end Main;

You can't specify both Add_Named and Add_Unnamed subprograms for the same type. This prevents you from defining JSON_Value with both array and object aggregates present. But we can define conversion functions for array and object and get code almost as dense as the same code in native JSON. For example:

Listing 4: main.adb

pragma Ada_2022;

procedure Main is

package JSON is
  type JSON_Value is private
    with Integer_Literal => To_Value, String_Literal => To_Value;
  function To_Value (Text : String) return JSON_Value;
  function To_Value (Text : Wide_Wide_String) return JSON_Value;

  type JSON_Object is private
    with Aggregate => (Empty => New_JSON_Object,
                      Add_Named => Insert);
  function New_JSON_Object return JSON_Object;

  procedure Insert
    (Self : in out JSON_Object;
     Key : Wide_Wide_String;
     Value : JSON_Value) is null;
  function From_Object (Self : JSON_Object) return JSON_Value;

  type JSON_Array is private
    with Aggregate => (Empty => New_JSON_Array,
                       Add_Unnamed => Append);
  function New_JSON_Array return JSON_Array;
end JSON;
procedure Append
(Self : in out JSON_Array;
 Value : JSON_Value) is null;

function From_Array (Self : JSON_Array) return JSON_Value;

private

type JSON_Value is null record;
type JSON_Object is null record;
type JSON_Array is null record;

function To_Value (Text : String) return JSON_Value is
(null record);
function To_Value (Text : Wide_Wide_String) return JSON_Value is
(null record);
function New_JSON_Object return JSON_Object is
(null record);
function New_JSON_Array return JSON_Array is
(null record);
function From_Object (Self : JSON_Object) return JSON_Value is
(null record);
function From_Array (Self : JSON_Array) return JSON_Value is
(null record);
end JSON;

function "+" (X : JSON.JSON_Object) return JSON.JSON_Value
renames JSON.From_Object;
function "+" (X : JSON.JSON_Array) return JSON.JSON_Value
renames JSON.From_Array;

Offices : JSON.JSON_Array :=
[+"name" => "North American Office",
 "phones" => [1.877.787.4628,
 1.866.787.4232,
 1.212.620.7300],
 "email" => "info@adacore.com"],
[+"name" => "European Office",
 "phones" => [33 1 49 70 67 16,
 33 1 49 70 05 52],
 "email" => "info@adacore.com"];

begin
-- Equivalent old initialization code is too long to print it here
null;
end Main;

Build output

main.adb:61:04: warning: variable "Offices" is not referenced [-gnatwu] 

The Offices variable is supposed to contain this value:

["name" : "North American Office",
 "phones": [18777874628,
 18667874232,
 12126207300],
 "email" : "info@adacore.com"],
{"name" : "European Office",
 "phones": [33149706716,
 33149700552],

(continues on next page)
6.1 References

- ARM 4.3.5 Container Aggregates\(^{20}\)
- AI12-0212-1\(^{21}\)

---

\(^{20}\) http://www.ada-auth.org/standards/22aarm/html/AA-4-3-5.html
\(^{21}\) http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0212-1.TXT
NOTE: Delta aggregates are supported by

- GNAT Community Edition 2019
- GCC 9

Sometimes you need to create a copy of an object, but with a few modifications. Before Ada 2022, doing this involves a dummy object declaration or an aggregate with associations for each property. The dummy object approach doesn't work in contract aspects or when there are limited components. On the other hand, re-listing properties in an large aggregate can be very tedious and error-prone. So, in Ada 2022, you can use a delta aggregate instead.

### 7.1 Delta aggregate for records

The delta aggregate for a record type looks like this:

```ada
type Vector is record
   X, Y, Z : Float;
end record;

Point_1 : constant Vector := (X => 1.0, Y => 2.0, Z => 3.0);
Projection_1 : constant Vector := (Point_1 with delta Z => 0.0);
```

The more components you have, the more you will like the delta aggregate.

### 7.2 Delta aggregate for arrays

You can also use delta aggregates for arrays to change elements, but not bounds. Moreover, it only works for one-dimensional arrays of non-limited components.

```ada
type Vector_3D is array (1 .. 3) of Float;

Point_2 : constant Vector_3D := [1.0, 2.0, 3.0];
Projection_2 : constant Vector_3D := [Point_2 with delta 3 => 0.0];
```

You can use parentheses for array aggregates, but you can't use square brackets for record aggregates.

Here is the complete code snippet:
Listing 1: main.adb

```ada
pragma Ada_2022;
with Ada.Text_IO;
procedure Main is
    type Vector is record
        X, Y, Z : Float;
    end record;

    Point_1 : constant Vector := (X => 1.0, Y => 2.0, Z => 3.0);
    Projection_1 : constant Vector := (Point_1 with delta Z => 0.0);

    type Vector_3D is array (1 .. 3) of Float;

    Point_2 : constant Vector_3D := [1.0, 2.0, 3.0];
    Projection_2 : constant Vector_3D := [Point_2 with delta 3 => 0.0];

    begin
        Ada.Text_IO.Put (Float'Image (Projection_1.X));
        Ada.Text_IO.Put (Float'Image (Projection_1.Y));
        Ada.Text_IO.Put (Float'Image (Projection_1.Z));
        Ada.Text_IO.New_Line;
        Ada.Text_IO.Put (Float'Image (Projection_2 (1)));  
        Ada.Text_IO.Put (Float'Image (Projection_2 (2)));  
        Ada.Text_IO.Put (Float'Image (Projection_2 (3)));  
        Ada.Text_IO.New_Line;
    end Main;
```

Runtime output

```
1.00000E+00 2.00000E+00 0.00000E+00
1.00000E+00 2.00000E+00 0.00000E+00
```

7.3 References

- ARM 4.3.4 Delta Aggregates\(^{22}\)
- AI12-0127-1\(^{23}\)

\(^{22}\) [http://www.ada-auth.org/standards/22aarm/html/AA-4-3-4.html](http://www.ada-auth.org/standards/22aarm/html/AA-4-3-4.html)
\(^{23}\) [http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0127-1.TXT](http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0127-1.TXT)
CHAPTER EIGHT

TARGET NAME SYMBOL (@)

Note: Target name symbol is supported by
  • GNAT Community Edition 2019
  • GCC 9

Ada 2022 introduces a new symbol, @, which can only appear on the right hand side of an assignment statement. This symbol acts as the equivalent of the name on the left hand side of that assignment statement. It was introduced to avoid code duplication: instead of retyping a (potentially long) name, you can use @. This symbol denotes a constant, so you can’t pass it into [in] out arguments of a subprogram.

As an example, let's calculate some statistics for My_Data array:

Listing 1: statistics.ads

```ada
pragma Ada_2022;

package Statistics is

  type Statistic is record
    Count : Natural := 0;
    Total : Float := 0.0;
  end record;

  My_Data : array (1 .. 5) of Float := [for J in 1 .. 5 => Float (J)];

 Statistic_For_My_Data : Statistic;

end Statistics;
```

To do this, we loop over My_Data elements:

Listing 2: main.adb

```ada
pragma Ada_2022;
with Ada.Text_IO;

procedure Main is

  type Statistic is record
    Count : Natural := 0;
    Total : Float := 0.0;
  end record;

  My_Data : constant array (1 .. 5) of Float :=
    [for J in 1 .. 5 => Float (J)];
```

(continues on next page)
What's New in Ada 2022

(continued from previous page)

```ada
Statistic_For_My_Data : Statistic;

begin
  for Data of My_Data loop
    Statistic_For_My_Data.Count := @ + 1;
    Statistic_For_My_Data.Total := @ + Data;
  end loop;
  Ada.Text_IO.Put_Line (Statistic_For_My_Data'Image);
end Main;
```

Runtime output

```
(COUNT => 5,
  TOTAL => 1.50000E+01)
```

Each right hand side is evaluated only once, no matter how many @ symbols it contains. Let's verify this by introducing a function call that prints a line each time it's called:

```
pragma Ada_2022;
with Ada.Text_IO;

procedure Main is
  My_Data : array (1 .. 5) of Float := [for J in 1 .. 5 => Float (J)];
  function To_Index (Value : Positive) return Positive is
    begin
      Ada.Text_IO.Put_Line ("To_Index is called.");
      return Value;
    end To_Index;
  begin
    My_Data (To_Index (1)) := @ ** 2 - 3.0 * @;
    Ada.Text_IO.Put_Line (My_Data'Image);
end Main;
```

Runtime output

```
To_Index is called.
[-2.0000E+00, 2.00000E+00, 3.00000E+00, 4.00000E+00, 5.00000E+00]
```

This use of @ may look a bit cryptic, but it's the best solution that was found. Unlike other languages (e.g., sum += x; in C), this approach lets you use @ an arbitrary number of times within the right hand side of an assignment statement.
8.1 Alternatives

In C++, the previous statement could be written with a reference type (one line longer!):

```cpp
auto a = my_data[to_index(1)];
a = a * a - 3.0 * a;
```

In Ada 2022, you can use a similar renaming:

```ada
declare
   A renames My_Data (To_Index (1));
begin
   A := A ** 2 - 3.0 * A;
end;
```

Here we use a new short form of therename declaration, but this still looks too heavy, and even worse, it can't be used for discriminant-dependent components.

8.2 References

- ARM 5.2.1 Target Name Symbols
- AI12-0125-3

25 http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0125-3.TXT
Note: Enumeration representation attributes are supported by

- GNAT Community Edition 2019
- GCC 9

Enumeration types in Ada are represented as integers at the machine level. But there are actually two mappings from enumeration to integer: a literal position and a representation value.

### 9.1 Literal positions

Each enumeration literal has a corresponding position in the type declaration. We can easily obtain it from the `Type'Pos (Enum)` attribute.

```
with Ada.Text_IO;  
with Ada.Integer_Text_IO;  

procedure Main is  
begin  
   Ada.Text_IO.Put ("Pos(False) =");  
   Ada.Integer_Text_IO.Put (Boolean'Pos (False));  
   Ada.Text_IO.New_Line;  
   Ada.Text_IO.Put ("Pos(True) =");  
   Ada.Integer_Text_IO.Put (Boolean'Pos (True));  
end Main;  
```

**Runtime output**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pos(False)</td>
<td>0</td>
</tr>
<tr>
<td>Pos(True)</td>
<td>1</td>
</tr>
</tbody>
</table>

For the reverse mapping, we use `Type'Val (Int)`:

```
with Ada.Text_IO;  

procedure Main is  
begin  
   Ada.Text_IO.Put_Line (Boolean'Val (0)'Image);  
   Ada.Text_IO.Put_Line (Boolean'Val (1)'Image);  
end Main;  
```
9.2 Representation values

The representation value defines the *internal* code, used to store enumeration values in memory or CPU registers. By default, enumeration representation values are the same as the corresponding literal positions, but you can redefine them. Here, we created a copy of `Boolean` type and assigned it a custom representation.

In Ada 2022, we can get an integer value of the representation with `Type'Enum_Rep(Enum)` attribute:

```ada
with Ada.Text_IO; with Ada.Integer_Text_IO;
procedure Main is
  type My_Boolean is new Boolean;
  for My_Boolean use (False => 3, True => 6);
begin
  Ada.Text_IO.Put ("Enum_Rep(False) =");
  Ada.Integer_Text_IO.Put (My_Boolean'Enum_Rep(False));
  Ada.Text_IO.Put_Line ("\n");
  Ada.Text_IO.Put ("Enum_Rep(True) =");
  Ada.Integer_Text_IO.Put (My_Boolean'Enum_Rep(True));
end Main;
```

**Runtime output**

```
Enum_Rep(False) = 3
Enum_Rep(True) = 6
```

And, for the reverse mapping, we can use `Type'Enum_Val (Int)`:  

```ada
with Ada.Text_IO; with Ada.Integer_Text_IO;
procedure Main is
  type My_Boolean is new Boolean;
  for My_Boolean use (False => 3, True => 6);
begin
  Ada.Text_IO.Put_Line (My_Boolean'Enum_Val(3)'Image);
  Ada.Text_IO.Put_Line (My_Boolean'Enum_Val(6)'Image);
  Ada.Text_IO.Put ("Pos(False) =");
  Ada.Integer_Text_IO.Put (My_Boolean'Pos(False));
  Ada.Text_IO.New_Line;
  Ada.Text_IO.Put ("Pos(True) =");
  Ada.Integer_Text_IO.Put (My_Boolean'Pos(True));
end Main;
```

**Runtime output**

```
Pos(False) = 3
Pos(True) = 6
```
FALSE
TRUE
Pos(False) = 0
Pos(True) = 1

Note that the \( \text{Val}(X)/\text{Pos}(X) \) behaviour still is the same.

Custom representations can be useful for integration with a low level protocol or hardware.

### 9.3 Before Ada 2022

This doesn't initially look like an important feature, but let's see how we'd do the equivalent with Ada 2012 and earlier versions. First, we need an integer type of matching size, then we instantiate Ada.Unchecked_Conversion. Next, we call To_Int/From_Int to work with representation values. And finally an extra type conversion is needed:

```
Listing 5: main.adb
```

```ada
with Ada.Text_IO;
with Ada.Integer_Text_IO;
with Ada.Unchecked_Conversion;

procedure Main is

  type My_Boolean is new Boolean;
  for My_Boolean use (False => 3, True => 6);
  type My_Boolean_Int is range 3 .. 6;
  for My_Boolean_Int'Size use My_Boolean'Size;

  function To_Int is new Ada.Unchecked_Conversion
          (My_Boolean, My_Boolean_Int);
  function From_Int is new Ada.Unchecked_Conversion
          (My_Boolean_Int, My_Boolean);

  begin
    Ada.Text_IO.Put ("To_Int(False) =");
    Ada.Integer_Text_IO.Put (Integer (To_Int (False)));
    Ada.Text_IO.New_Line;
    Ada.Text_IO.Put ("To_Int(True) =");
    Ada.Integer_Text_IO.Put (Integer (To_Int (True)));
    Ada.Text_IO.New_Line;
    Ada.Text_IO.Put ("From_Int (3) =");
    Ada.Text_IO.Put_Line (From_Int (3)'Image);
    Ada.Text_IO.New_Line;
    Ada.Text_IO.Put ("From_Int (6) =");
    Ada.Text_IO.Put_Line (From_Int (6)'Image);
  end Main;
```

**Runtime output**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>To_Int(False)</td>
<td>3</td>
</tr>
<tr>
<td>To_Int(True)</td>
<td>6</td>
</tr>
<tr>
<td>From_Int (3)</td>
<td>TRUE</td>
</tr>
<tr>
<td>From_Int (6)</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

Even with all that, this solution doesn't work for generic formal type (because T'Size must be a static value)!
We should note that these new attributes may already be familiar to GNAT users because they've been in the GNAT compiler for many years.

9.4 References

- ARM 13.4 Enumeration Representation Clauses\textsuperscript{26}
- AI12-0237-1\textsuperscript{27}

\textsuperscript{26} [Link to ARM 13.4 Enumeration Representation Clauses]
\textsuperscript{27} [Link to AI12-0237-1]
Note: Big numbers are supported by

- GNAT Community Edition 2020
- GCC 11
- GCC 10 (draft, no user defined literals)

Ada 2022 introduces big integers and big real types.

10.1 Big Integers

The package `Ada.Numerics.Big_Numbers.Big_Integers` contains a type `Big_Integer` and corresponding operations such as comparison (=, <, >, <=, >=), arithmetic (+, -, *, /, rem, mod, abs, **), Min, Max and Greatest_Common_Divisor. The type also has Integer_Literal and Put_Image aspects redefined, so you can use it in a natural manner.

```ada
Ada.Text_IO.Put_Line (Big_Integer'Image(2 ** 256));
```

115792089237316195423570985008687907853269984665640564039457584007913129639936

10.2 Tiny RSA implementation

Note: Note that you shouldn't use Big_Numbers for cryptography because it's vulnerable to timing side-channels attacks.

We can implement the RSA algorithm :wikipedia:`RSA algorithm <RSA_(cryptosystem)>` in a few lines of code. The main operation of RSA is \( m^d \mod n \). But you can't just write \( m^d \), because these are really big numbers and the result won't fit into memory. However, if you keep intermediate result \( m^d \mod n \) during the \( m^d \) calculation, it will work. Let's write this operation as a function:

```
Listing 1: power_mod.ads
pragma Ada_2022;
with Ada.Numerics.Big_Numbers.Big_Integers;
use Ada.Numerics.Big_Numbers.Big_Integers;
```
--- Calculate $M^D \mod N$

function Power_Mod (M, D, N : Big_Integer) return Big_Integer;

Listing 2: power_mod.adb

function Power_Mod (M, D, N : Big_Integer) return Big_Integer is
  function Is_Odd (X : Big_Integer) return Boolean is
    (X mod 2 /= 0);
  begin
    Result := 1;
    Exp := D;
    Mult := M mod N;
    while Exp /= 0 loop
      if Is_Odd (Exp) then
        Result := (Result * Mult) mod N;
      end if;
      Mult := Mult \* 2 mod N;
      Exp := Exp / 2;
    end loop;
    return Result;
  end Power_Mod;

Let's check this with the example from Wikipedia\(^\text{28}\). In that example, the public key is $(n = 3233, e = 17)$ and the message is $m = 65$. The encrypted message is $m^e \mod n = 65^{17} \mod 3233 = 2790 = c$.

Ada.Text_IO.Put_Line (Power_Mod (M => 65, D => 17, N => 3233)'Image);

2790

To decrypt it with the public key $(n = 3233, d = 413)$, we need to calculate $c^d \mod n = 2790^{413} \mod 3233$:

Ada.Text_IO.Put_Line (Power_Mod (M => 2790, D => 413, N => 3233)'Image);

65

So 65 is the original message $m$. Easy!

Here is the complete code snippet:

Listing 3: main.adb

pragma Ada_2022;

with Ada.Text_IO;
with Ada.Numerics.Big_Numbers.Big_Integers;
use Ada.Numerics.Big_Numbers.Big_Integers;

procedure Main is
  -- Calculate $M^D \mod N$

\(^{28}\) https://en.wikipedia.org/wiki/RSA_(cryptosystem)
function Power_Mod (M, D, N : Big_Integer) return Big_Integer is

function Is_Odd (X : Big_Integer) return Boolean is
(X mod 2 /= 0);

Result : Big_Integer := 1;
Exp : Big_Integer := D;
Mult : Big_Integer := M mod N;

begin
while Exp /= 0 loop
  -- Loop invariant is Power_Mod'Result = Result * Mult**Exp mod N
  if Is_Odd (Exp) then
    Result := (Result * Mult) mod N;
  end if;
  Mult := Mult * 2 mod N;
  Exp := Exp / 2;
end loop;

return Result;
end Power_Mod;

begin
  Ada.Text_IO.Put_Line (Big_Integer'Image (2 ** 256));
  Ada.Text_IO.Put_Line (Power_Mod (M => 65, D => 17, N => 3233)'Image);
end Main;

Runtime output

1157920892373161954235709895008687907853269984665640564039457584007913129639936
2790
65

10.3 Big Reals

In addition to Big_Integer, Ada 2022 provides Big Reals. In addition to Big_Integer, Ada 2022 provides Big Reals. In addition to Big_Integer, Ada 2022 provides Big Reals.

10.4 References

- ARM A.5.6 Big Integers
- ARM A.5.7 Big Reals
- A12-0208-1
CHAPTER
ELEVEN

INTERFACING C VARIADIC FUNCTIONS

Note: Variadic convention is supported by

- GNAT Community Edition 2020
- GCC 11

In C, variadic functions\(^{33}\) take a variable number of arguments and an ellipsis as the last parameter of the declaration. A typical and well-known example is:

```c
int printf (const char* format, ...);
```

Usually, in Ada, we bind such a function with just the parameters we want to use:

```ada
procedure printf_double
  (format : Interfaces.C.char_array;
   value : Interfaces.C.double)
with Import,
   Convention => C,
   External_Name => "printf";
```

Then we call it as a normal Ada function:

```ada
printf_double (Interfaces.C.To_C ("Pi=%f"), Ada.Numerics.π);
```

Unfortunately, doing it this way doesn't always work because some ABI\(^{34}\)'s use different calling conventions for variadic functions. For example, the AMD64 ABI\(^{35}\) specifies:

- `%rax` — with variable arguments passes information about the number of vector registers used;
- `%xmm0–%xmm1` — used to pass and return floating point arguments.

This means, if we write (in C):

```c
printf("%d", 5);
```

The compiler will place 0 into `%rax`, because we don’t pass any float argument. But in Ada, if we write:

```ada
procedure printf_int
  (format : Interfaces.C.char_array;
   value : Interfaces.C.int)
with Import,
   Convention => C,
      (continues on next page)
```

\(^{33}\) https://en.cppreference.com/w/c/variadic

\(^{34}\) https://en.wikipedia.org/wiki/Application_binary_interface

What's New in Ada 2022

(continued from previous page)

```ada
    External_Name => "printf";

    printf_int (Interfaces.C.To_C ("d=%d"), 5);
```

the compiler won't use the %rax register at all. (You can't include any float argument because there's no float parameter in the Ada wrapper function declaration.) As result, you will get a crash, stack corruption, or other undefined behavior.

To fix this, Ada 2022 provides a new family of calling convention names — C_Variadic_N:

    The convention C_Variadic_n is the calling convention for a variadic C function taking n fixed parameters and then a variable number of additional parameters.

Therefore, the correct way to bind the printf function is:

```ada
procedure printf_int
  (format : Interfaces.C.char_array;
   value : Interfaces.C.int)
with Import,
  Convention => C_Variadic_1,
  External_Name => "printf";
```

And the following call won't crash on any supported platform:

```ada
printf_int (Interfaces.C.To_C ("d=%d"), 5);
```

Without this convention, problems cause by this mismatch can be very hard to debug. So, this is a very useful extension to the Ada-to-C interfacing facility.

Here is the complete code snippet:

Listing 1: main.adb

```ada
with Interfaces.C;

procedure Main is
  procedure printf_int
    (format : Interfaces.C.char_array;
     value : Interfaces.C.int)
  with Import,
    Convention => C_Variadic_1,
    External_Name => "printf";

begin
  printf_int (Interfaces.C.To_C ("d=%d"), 5);
end Main;
```

### 11.1 References

- ARM B.3 Interfacing with C and C++\(^{36}\)
- AI12-0028-1\(^{37}\)

---


\(^{37}\) [http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0028-1.TXT](http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0028-1.TXT)