What’s New in Ada 2022
Maxim Reznik
What's New in Ada 2022

Release 2024-06

Maxim Reznik

Jun 30, 2024
CONTENTS:

1 Introduction
   1.1 References ................................................. 3

2 'Image attribute for any type
   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
   3.1 What's the Root_Buffer_Type? .............................. 8
   3.2 Outdated draft implementation ............................ 8
   3.3 References ................................................. 8

4 User-Defined Literals
   4.1 Turn Ada into JavaScript .................................. 12
   4.2 References ................................................. 13

5 Advanced Array Aggregates
   5.1 Square brackets .......................................... 15
   5.2 Iterated Component Association .......................... 16
   5.3 References ................................................. 17

6 Container Aggregates
   6.1 References ................................................. 19

7 Delta Aggregates
   7.1 Delta aggregate for records .............................. 25
   7.2 Delta aggregate for arrays ............................... 25
   7.3 References ................................................. 26

8 Target Name Symbol (@)
   8.1 Alternatives .............................................. 27
   8.2 References ................................................. 29

9 Enumeration representation
   9.1 Literal positions .......................................... 31
   9.2 Representation values .................................... 32
   9.3 Before Ada 2022 .......................................... 33
   9.4 References ................................................. 34

10 Big Numbers
   10.1 Big Integers .............................................. 35
   10.2 Tiny RSA implementation ................................ 35
   10.3 Big Reals .................................................. 37
   10.4 References ................................................. 38
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.

Note: The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

Note: Each code example from this book has an associated "code block metadata", which contains the name of the "project" and an MD5 hash value. This information is used to identify a single code example.

You can find all code examples in a zip file, which you can download from the learn website. The directory structure in the zip file is based on the code block metadata. For example, if you're searching for a code example with this metadata:

- Project: Courses.Intro_To_Ada.Imperative_Language.Greet
- MD5: cba89a34b87c9dfa71533d982d05e6ab

you will find it in this directory:

```
projects/Courses/Intro_To_Ada/Imperative_Language/Greet/cba89a34b87c9dfa71533d982d05e6ab/
```

In order to use this code example, just follow these steps:

1. Unpack the zip file;
2. Go to target directory;
3. Start GNAT Studio on this directory;
4. Build (or compile) the project;
5. Run the application (if a main procedure is available in the project).

---

1 http://creativecommons.org/licenses/by-sa/4.0
2 https://learn.adacore.com/zip/learning-ada_code.zip
INTRODUCTION

This is a collection of short code examples demonstrating new features of the Ada 2022 Standard\(^3\) 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\(^4\) or GCC 11\(^5\) 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\(^6\)
- Ada 202x support in GNAT\(^7\) blog post

---

\(^4\) [https://blog.adacore.com/gnat-community-2021-is-here](https://blog.adacore.com/gnat-community-2021-is-here)
\(^7\) [https://blog.adacore.com/ada-202x-support-in-gnat](https://blog.adacore.com/ada-202x-support-in-gnat)
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 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. These two statements are equivalent:

```
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;
```

---

8 https://reznikmm.github.io/ada-auth/rm-4-NC/RM-0-1.html
9 https://reznikmm.github.io/ada-auth/rm-4-NC/RM-4-1.html#S0091
What's New in Ada 2022

(continued from previous page)

```
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;
```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Image_Attribute
MD5: 47945f0f8a4ba37b838f87b7e5acaa49

Runtime output

```
[ 1, 2, 3]
(LINE => 10,
 COLUMN => 3)
(access 7ffdd1e99348)
```

```
$ 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 15)!

## 2.3 References

- ARM 4.10 Image Attributes\(^\text{10}\)
- AI12-0020-1\(^\text{11}\)

---


\(^\text{11}\) [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

(continued from previous page)

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

### Code block metadata

MD5: a4a6df87eaa66d0a2bcaac9c4cccbe4a

### 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.

```ada
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:

```ada
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\(^\text{12}\)
- AI12-0020-1\(^\text{13}\)
- AI12-0384-2\(^\text{14}\)

\(^{13}\) [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)
\(^{14}\) [http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/AI12-0384-2.TXT](http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/AI12-0384-2.TXT)
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)
begin
  Print ("Test "string"\n);  
  Print (123);  
  Print (16#DEAD_BEEF#);  
  Print (2.99_792_458e+8);  
end Main;

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?

```ada
> '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

```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 "+" (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)));
```
What's New in Ada 2022

(continued from previous page)

```ada
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;
```

Code block metadata

MD5: 9f41f61b1f4bc03cbe245cd8e0288e4f

Runtime output

53

Jokes aside, this feature is very useful. For example it allows a "native-looking API" for big integers (page 35).

4.2 References

- ARM 4.2.1 User-Defined Literals
- AI12-0249-1
- AI12-0342-1

15 http://www.ada-auth.org/standards/22rm/html/RM-4-2-1.html
16 http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0249-1.TXT
17 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;
```

Code block metadata

Project: Courses.Ada 2022 Whats New.Square_Brackets
MD5: fb4638717d4a12c1da8e646705ddf17

Short summary for parentheses and brackets

- Record aggregates use parentheses
- Container aggregates (page 19) 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

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;

5.3 References

- ARM 4.3.3 Array Aggregates\(^{18}\)
- AI12-0212-1\(^{19}\)
- AI12-0306-1\(^{20}\)

---

\(^{18}\) http://www.ada-auth.org/standards/22aarm/html/AA-4-3-3.html
\(^{19}\) http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0212-1.TXT
\(^{20}\) http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0306-1.TXT
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;
```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Container_Aggregates_1
MD5: dd1dd78890d4bf6c78b79d56abba332d

Runtime output

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

```
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:

```
pragma Ada_2022;

procedure Main is

package JSON is
  type JSON_Value is private
```

(continues on next page)
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,
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;

private

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;

Code block metadata

MD5: 758ced718aa9a4eefa32325543eb3b1e

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,
                      AddUnnamed => Append);

function New_JSON_Array return JSON_Array;

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
  null;
end Main;
The Offices variable is supposed to contain this value:

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

### 6.1 References

- ARM 4.3.5 Container Aggregates\(^\text{21}\)
- AI12-0212-1\(^\text{22}\)

---

\(^{21}\) [http://www.ada-auth.org/standards/22aarm/html/AA-4-3-5.html](http://www.ada-auth.org/standards/22aarm/html/AA-4-3-5.html)
\(^{22}\) [http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0212-1.TXT](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

```
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;
```

7.3 References

- ARM 4.3.4 Delta Aggregates\(^{23}\)
- AI12-0127-1\(^ {24}\)


\(^{24}\) [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)
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)];

end Statistics;
```

**Code block metadata**

Project: Courses.Ada_2022_Whats_New.Assignment_Tagged_Intro
MD5: 5cc813a4a22d3acc8418b0c1c6df3877

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;

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

(continued from previous page)

```ada
Total : Float := 0.0;
end record;

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

Code block metadata

Project: Courses.Ada_2022_Whats_New.Assignment_Tagged_2
MD5: 10dd019f4c09bc950895a93b3a88b778

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:

Listing 3: main.adb

```ada
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;
```

Code block metadata

MD5: 98d6afbaee5c0f6cd2bebe6b39962ad3

Runtime output

To_Index is called.
[-2.00000E+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 the rename 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

---

26 [http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0125-3.TXT](http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0125-3.TXT)
CHAPTER NINE

ENUMERATION REPRESENTATION

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.

Listing 1: main.adb

```ada
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;
```

For the reverse mapping, we use Type'Val (Int):
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.New_Line;  
  Ada.Text_IO.Put("Enum_Rep(True) =");  
  Ada.Integer_Text_IO.Put(My_Boolean'Enum_Rep(True));  
end Main;  
```

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  
begin  
  Ada.Text_IO.Put("Enum_Rep(False) =");  
  Ada.Integer_Text_IO.Put(Enum_Rep(False));  
  Ada.Text_IO.New_Line;  
  Ada.Text_IO.Put("Enum_Rep(True) =");  
  Ada.Integer_Text_IO.Put(Enum_Rep(True));  
end Main;  
```
What's New in Ada 2022

Listing 4: main.adb

```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_Line ("Pos(False) = ");
  Ada.Integer_Text_IO.Put (My_Boolean'Pos (False));
  Ada.Text_IO.New_Line;
  Ada.Text_IO.Put_Line ("Pos(True) = ");
  Ada.Integer_Text_IO.Put (My_Boolean'Pos (True));
end Main;
```

Runtime output

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

Note that the 'Val(X)'/'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
```

9.3. Before Ada 2022
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;

Code block metadata
MD5: 7c7624ed024b26036389f77dbd6cb109

Runtime output

<table>
<thead>
<tr>
<th>Description</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\(^{27}\)
- AI12-0237-1\(^{28}\)

\(^{28}\) [http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0237-1.TXT](http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0237-1.TXT)
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.Text_IO.Put_Line (Big_Integer'Image (2 ** 256));

1157920892316195423570985008687907853269984665640564039457584007913129639936

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

Listing 1: power_mod.ads

```ada
pragma Ada_2022;
with Ada.Numerics.Big_Numbers.Big_Integers;
use Ada.Numerics.Big_Numbers.Big_Integers;
(continues on next page)
```

-- 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
      -- 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;

Code block metadata

Project: Courses.Ada_2022_Whats_New.Big_Integers
MD5: 217c2aa3535952b68e2f088d262e6f60

Let's check this with the example from Wikipedia\(^{30}\). 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;  

\(^{30}\) https://en.wikipedia.org/wiki/RSA_(cryptosystem)
with Ada.Numerics.Big_Numbers.Big_Integers;
use Ada.Numerics.Big_Numbers.Big_Integers;

procedure Main is
  -- Calculate M ** D mod N

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

10.3 Big Reals

In addition to Big_Integer, Ada 2022 provides Big Reals\(^{31}\).

10.4 References

- ARM A.5.6 Big Integers\textsuperscript{32}
- ARM A.5.7 Big Reals\textsuperscript{33}
- AI12-0208-1\textsuperscript{34}

\textsuperscript{34} http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0208-1.TXT
INTERFACING C VARIADIC FUNCTIONS

Note: Variadic convention is supported by

- GNAT Community Edition 2020
- GCC 11

In C, variadic functions\(^{35}\) 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.\pi);
```

Unfortunately, doing it this way doesn't always work because some ABIs\(^{36}\) use different calling conventions for variadic functions. For example, the AMD64 ABI\(^{37}\) 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)

---

\(^{35}\) https://en.cppreference.com/w/c/variadic

\(^{36}\) https://en.wikipedia.org/wiki/Application_binary_interface

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:

```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;
```

**Code block metadata**

MD5: 94515f55a93f27e4f4ec3c31256645d9
11.1 References

- ARM B.3 Interfacing with C and C++.\(^{38}\)
- AI12-0028-1\(^{39}\)

\(^{39}\) http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0028-1.TXT