

# What's New in Ada 2022

Maxim Reznik

**LEARN.**  
ADACORE.COM

# **What's New in Ada 2022**

*Release 2024-09*

**Maxim Reznik**

**Sep 08, 2024**



## CONTENTS:

<b>1 Introduction</b>	<b>3</b>
1.1 References . . . . .	3
<b>2 'Image attribute for any type</b>	<b>5</b>
2.1 'Image attribute for a value . . . . .	5
2.2 'Image attribute for any type . . . . .	5
2.3 References . . . . .	6
<b>3 Redefining the 'Image attribute</b>	<b>7</b>
3.1 What's the Root_Buffer_Type? . . . . .	8
3.2 Outdated draft implementation . . . . .	8
3.3 References . . . . .	9
<b>4 User-Defined Literals</b>	<b>11</b>
4.1 Turn Ada into JavaScript . . . . .	12
4.2 References . . . . .	13
<b>5 Advanced Array Aggregates</b>	<b>15</b>
5.1 Square brackets . . . . .	15
5.2 Iterated Component Association . . . . .	16
5.3 References . . . . .	17
<b>6 Container Aggregates</b>	<b>19</b>
6.1 References . . . . .	23
<b>7 Delta Aggregates</b>	<b>25</b>
7.1 Delta aggregate for records . . . . .	25
7.2 Delta aggregate for arrays . . . . .	25
7.3 References . . . . .	26
<b>8 Target Name Symbol (@)</b>	<b>27</b>
8.1 Alternatives . . . . .	29
8.2 References . . . . .	29
<b>9 Enumeration representation</b>	<b>31</b>
9.1 Literal positions . . . . .	31
9.2 Representation values . . . . .	32
9.3 Before Ada 2022 . . . . .	33
9.4 References . . . . .	34
<b>10 Big Numbers</b>	<b>35</b>
10.1 Big Integers . . . . .	35
10.2 Tiny RSA implementation . . . . .	35
10.3 Big Reals . . . . .	38
10.4 References . . . . .	38

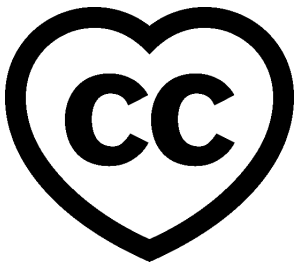
<b>11 Interfacing C variadic functions</b>	<b>39</b>
11.1 References . . . . .	41

**Warning**

This version of the website contains UNPUBLISHED contents. Please do not share it externally!

Copyright © 2022, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page](#)<sup>1</sup>



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](#)<sup>2</sup>. 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;

<sup>1</sup> <http://creativecommons.org/licenses/by-sa/4.0>

4. Build (or compile) the project;
5. Run the application (if a main procedure is available in the project).

---

<sup>2</sup> [https://learn.adacore.com/zip/learning-ada\\_code.zip](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](#)<sup>3</sup> 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](#)<sup>4</sup> or [GCC 11](#)<sup>5</sup> 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](#)<sup>6</sup>
- [Ada 202x support in GNAT](#)<sup>7</sup> blog post

---

<sup>3</sup> <http://www.ada-auth.org/standards/22aarm/html/AA-TTL.html>

<sup>4</sup> <https://blog.adacore.com/gnat-community-2021-is-here>

<sup>5</sup> <https://gcc.gnu.org/gcc-11/>

<sup>6</sup> <http://www.ada-auth.org/standards/22aarm/html/AA-TTL.html>

<sup>7</sup> <https://blog.adacore.com/ada-202x-support-in-gnat>





## '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<sup>8</sup>](#) 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`<sup>9</sup>. These two statements are equivalent:

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

## 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

```
1 pragma Ada_2022;  
2  
3 with Ada.Text_IO;  
4  
5 procedure Main is  
6   type Vector is array (Positive range <>) of Integer;  
7  
8   V1 : aliased Vector := [1, 2, 3];  
9  
10  type Text_Position is record
```

(continues on next page)

<sup>8</sup> <https://reznikmm.github.io/ada-auth/rm-4-NC/RM-0-1.html>

<sup>9</sup> <https://reznikmm.github.io/ada-auth/rm-4-NC/RM-4-1.html#S0091>

(continued from previous page)

```
11     Line, Column : Positive;  
12 end record;  
13  
14 Pos : constant Text_Position := (Line => 10, Column => 3);  
15  
16 type Vector_Access is access all Vector;  
17  
18 V1_Ptr : constant Vector_Access := V1'Access;  
19  
20 begin  
21   Ada.Text_IO.Put_Line (V1'Image);  
22   Ada.Text_IO.Put_Line (Pos'Image);  
23   Ada.Text_IO.New_Line;  
24   Ada.Text_IO.Put_Line (V1_Ptr'Image);  
25 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 7ffe67bd4c88)
```

```
$ 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<sup>10</sup>
- AI12-0020-1<sup>11</sup>

---

<sup>10</sup> <http://www.ada-auth.org/standards/22aarm/html/AA-4-10.html>

<sup>11</sup> <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)
- GNAT Community Edition 2020 (using Text\_Output.Utils)
- 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

```
1 pragma Ada_2022;
2
3 with Ada.Text_IO;
4 with Ada.Strings.Text_Buffers;
5
6 procedure Main is
7
8   type Source_Location is record
9     Line   : Positive;
10    Column : Positive;
11  end record
12    with Put_Image => My_Put_Image;
13
14  procedure My_Put_Image
15    (Output : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
16     Value  : Source_Location);
17
18  procedure My_Put_Image
19    (Output : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
20     Value  : Source_Location)
21  is
22    Line   : constant String := Value.Line'Image;
23    Column : constant String := Value.Column'Image;
24    Result : constant String :=
25      Line (2 .. Line'Last) & ':' & Column (2 .. Column'Last);
26  begin
27    Output.Put (Result);
28  end My_Put_Image;
29
30  Line_10 : constant Source_Location := (Line => 10, Column => 1);
```

(continues on next page)

(continued from previous page)

```
31
32 begin
33     Ada.Text_IO.Put_Line (Line_10'Image);
34 end Main;
```

### Code block metadata

```
Project: Courses.Ada_2022_Whats_New.Image_Redefine
MD5: a4a6df87eea66d0a2bcaac9c4cccbe4a
```

### 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](#)<sup>12</sup>
- [AI12-0020-1](#)<sup>13</sup>
- [AI12-0384-2](#)<sup>14</sup>

---

<sup>12</sup> <http://www.ada-auth.org/standards/22aarm/html/AA-4-10.html>

<sup>13</sup> <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0020-1.TXT>

<sup>14</sup> <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/AI12-0384-2.TXT>



## 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

```
1 pragma Ada_2022;
2
3 with Ada.Wide_Wide_Text_IO;
4 with Ada.Characters.Conversions;
5
6 procedure Main is
7
8   type My_Type (Length : Natural) is record
9     Value : Wide_Wide_String (1 .. Length);
10  end record
11   with String_Literal => From_String,
12        Real_Literal   => From_Real,
13        Integer_Literal => From_Integer;
14
15  function From_String (Value : Wide_Wide_String) return My_Type is
16    ((Length => Value'Length, Value => Value));
17
18  function From_Real (Value : String) return My_Type is
19    ((Length => Value'Length,
20     Value => Ada.Characters.Conversions.To_Wide_Wide_String (Value)));
21
22  function From_Integer (Value : String) return My_Type renames From_Real;
23
24  procedure Print (Self : My_Type) is
25  begin
```

(continues on next page)



(continued from previous page)

```
26     Ada.Wide_Wide_Text_IO.Put_Line (Self.Value);
27     end Print;
28
29 begin
30     Print ("Test ""string""");
31     Print (123);
32     Print (16#DEAD_BEEF#);
33     Print (2.99_792_458e+8);
34 end Main;
```

### Code block metadata

Project: Courses.Ada\_2022\_Whats\_New.User\_Defined\_Literals  
MD5: 3a4a12aa148b6845a1130e818e16c405

### 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

```
1  pragma Ada_2022;
2
3  with Ada.Wide_Wide_Text_IO;
4  with Ada.Characters.Conversions;
5
6  procedure Main is
7
8      type My_Type (Length : Natural) is record
9          Value : Wide_Wide_String (1 .. Length);
10     end record
11     with String_Literal => From_String,
12          Real_Literal   => From_Real,
13          Integer_Literal => From_Integer;
14
15     function "+" (Left, Right : My_Type) return My_Type is
16         (Left.Length + Right.Length, Left.Value & Right.Value);
17
18     function From_String (Value : Wide_Wide_String) return My_Type is
19         ((Length => Value'Length, Value => Value));
20
```

(continues on next page)

(continued from previous page)

```

21  function From_Real (Value : String) return My_Type is
22      ((Length => Value'Length,
23       Value => Ada.Characters.Conversions.To_Wide_Wide_String (Value)));
24
25  function From_Integer (Value : String) return My_Type renames From_Real;
26
27  procedure Print (Self : My_Type) is
28  begin
29      Ada.Wide_Wide_Text_IO.Put_Line (Self.Value);
30  end Print;
31
32  begin
33      Print ("5" + 3);
34  end Main;

```

**Code block metadata**

Project: Courses.Ada\_2022\_Whats\_New.User\_Defined\_Literals\_JS  
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<sup>15</sup>
- AI12-0249-1<sup>16</sup>
- AI12-0342-1<sup>17</sup>

<sup>15</sup> <http://www.ada-auth.org/standards/22rm/html/RM-4-2-1.html>

<sup>16</sup> <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0249-1.TXT>

<sup>17</sup> <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0342-1.TXT>



## ADVANCED ARRAY AGGREGATES

### 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

```
1 pragma Ada_2022;  
2 pragma Extensions_Allowed (On);  
3  
4 package Show_Square_Brackets is  
5  
6     type Integer_Array is array (Positive range <>) of Integer;  
7  
8     Old_Style_Empty : Integer_Array := (1 .. 0 => <>);  
9     New_Style_Empty : Integer_Array := [];  
10  
11     Old_Style_One_Item : Integer_Array := (1 => 5);  
12     New_Style_One_Item : Integer_Array := [5];  
13  
14 end Show_Square_Brackets;
```

#### Code block metadata

Project: Courses.Ada\_2022\_Whats\_New.Square\_Brackets  
MD5: fb4638717d4a12c1dae8e646705ddf17

### 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

```
1 pragma Ada_2022;
2 pragma Extensions_Allowed (On); -- for square brackets
3
4 with Ada.Text_IO;
5
6 procedure Show_Iterated_Component_Association is
7
8   type Integer_Array is array (Positive range <>) of Integer;
9
10  Old_Style_Empty : Integer_Array := (1 .. 0 => <>);
11  New_Style_Empty : Integer_Array := [];
12
13  Old_Style_One_Item : Integer_Array := (1 => 5);
14  New_Style_One_Item : Integer_Array := [5];
15
16  Vector : constant Integer_Array := [for J in 1 .. 5 => J * 2];
17
18  Matrix : constant array (1 .. 3, 1 .. 3) of Positive :=
19    [for J in 1 .. 3 =>
20      [for K in 1 .. 3 => J * 10 + K]];
21
22  Data : constant Integer_Array (1 .. 5) :=
23    [for J in 2 .. 3 => J, 5 => 5, others => 0];
24
25  Vector_2 : constant Integer_Array := [for X of Vector => X / 2];
```

(continues on next page)

(continued from previous page)

```
26 begin
27   Ada.Text_IO.Put_Line (Vector'Image);
28   Ada.Text_IO.Put_Line (Matrix'Image);
29   Ada.Text_IO.Put_Line (Data'Image);
30   Ada.Text_IO.Put_Line (Vector_2'Image);
31 end Show_Iterated_Component_Association;
```

### Code block metadata

```
Project: Courses.Ada_2022_Whats_New.Iterated_Component_Association
MD5: 05f7fc94e3f4d79b7ca25de4d7dedf4f
```

### 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](#)<sup>18</sup>
- [AI12-0212-1](#)<sup>19</sup>
- [AI12-0306-1](#)<sup>20</sup>

<sup>18</sup> <http://www.ada-auth.org/standards/22aarm/html/AA-4-3-3.html>

<sup>19</sup> <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0212-1.TXT>

<sup>20</sup> <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0306-1.TXT>



## 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

```
1 pragma Ada_2022;
2
3 with Ada.Text_IO;
4 with Ada.Containers.Vectors;
5 with Ada.Containers.Ordered_Maps;
6
7 procedure Main is
8
9     package Int_Vectors is new Ada.Containers.Vectors
10        (Positive, Integer);
11
12     X : constant Int_Vectors.Vector := [1, 2, 3];
13
14     package Float_Maps is new Ada.Containers.Ordered_Maps
15        (Integer, Float);
16
17     Y : constant Float_Maps.Map := [-10 => 1.0, 0 => 2.5, 10 => 5.51];
18 begin
19     Ada.Text_IO.Put_Line (X'Image);
20     Ada.Text_IO.Put_Line (Y'Image);
21 end Main;
```

### Code block metadata

Project: Courses.Ada\_2022\_Whats\_New.Container\_Aggregates\_1  
MD5: dd1dd78890d4bf6c78b79d56abba332d

### Runtime output

```
[ 1,  2,  3]
```

(continues on next page)



(continued from previous page)

```
[-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:

Listing 2: main.adb

```
1 pragma Ada_2022;
2
3 procedure Main is
4
5   package JSON is
6     type JSON_Value is private
7       with Integer_Literal => To_JSON_Value;
8
9     function To_JSON_Value (Text : String) return JSON_Value;
10
11    type JSON_Array is private
12      with Aggregate => (Empty      => New_JSON_Array,
13                        Add_Unnamed => Append);
14
15    function New_JSON_Array return JSON_Array;
16
17    procedure Append
18      (Self : in out JSON_Array;
19       Value : JSON_Value) is null;
19
20
21   private
22     type JSON_Value is null record;
23     type JSON_Array is null record;
24
25     function To_JSON_Value (Text : String) return JSON_Value
26       is (null record);
27
28     function New_JSON_Array return JSON_Array is (null record);
29   end JSON;
30
31   List : JSON.JSON_Array := [1, 2, 3];
32   -----
33 begin
34   -- Equivalent old initialization code
35   List := JSON.New_JSON_Array;
36   JSON.Append (List, 1);
37   JSON.Append (List, 2);
38   JSON.Append (List, 3);
39 end Main;
```

### Code block metadata

```
Project: Courses.Ada_2022_Whats_New.Container_Aggregates_2
MD5: 9cf1fefa4a725083c50794146d5cbde7
```

The equivalent for maps is:

Listing 3: main.adb

```
1 pragma Ada_2022;
2
```

(continues on next page)

(continued from previous page)

```

3 procedure Main is
4
5   package JSON is
6     type JSON_Value is private
7       with Integer_Literal => To_JSON_Value;
8
9     function To_JSON_Value (Text : String) return JSON_Value;
10
11    type JSON_Object is private
12      with Aggregate => (Empty      => New_JSON_Object,
13                        Add_Named => Insert);
14
15    function New_JSON_Object return JSON_Object;
16
17    procedure Insert
18      (Self : in out JSON_Object;
19       Key  : Wide_Wide_String;
20       Value : JSON_Value) is null;
21
22  private
23    type JSON_Value is null record;
24    type JSON_Object is null record;
25
26    function To_JSON_Value (Text : String) return JSON_Value
27      is (null record);
28
29    function New_JSON_Object return JSON_Object is (null record);
30  end JSON;
31
32  Object : JSON.JSON_Object := ["a" => 1, "b" => 2, "c" => 3];
33  -----
34  begin
35    -- Equivalent old initialization code
36    Object := JSON.New_JSON_Object;
37    JSON.Insert (Object, "a", 1);
38    JSON.Insert (Object, "b", 2);
39    JSON.Insert (Object, "c", 3);
40  end Main;

```

**Code block metadata**

Project: Courses.Ada\_2022\_Whats\_New.Container\_Aggregates\_3  
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

```

1 pragma Ada_2022;
2
3 procedure Main is
4
5   package JSON is
6     type JSON_Value is private
7       with Integer_Literal => To_Value, String_Literal => To_Value;
8
9     function To_Value (Text : String) return JSON_Value;
10    function To_Value (Text : Wide_Wide_String) return JSON_Value;

```

(continues on next page)

(continued from previous page)

```

11
12  type JSON_Object is private
13     with Aggregate => (Empty      => New_JSON_Object,
14                       Add_Named => Insert);
15
16  function New_JSON_Object return JSON_Object;
17
18  procedure Insert
19     (Self : in out JSON_Object;
20      Key  : Wide_Wide_String;
21      Value : JSON_Value) is null;
22
23  function From_Object (Self : JSON_Object) return JSON_Value;
24
25  type JSON_Array is private
26     with Aggregate => (Empty      => New_JSON_Array,
27                       Add_Unnamed => Append);
28
29  function New_JSON_Array return JSON_Array;
30
31  procedure Append
32     (Self : in out JSON_Array;
33      Value : JSON_Value) is null;
34
35  function From_Array (Self : JSON_Array) return JSON_Value;
36
37  private
38  type JSON_Value is null record;
39  type JSON_Object is null record;
40  type JSON_Array is null record;
41
42  function To_Value (Text : String) return JSON_Value is
43     (null record);
44  function To_Value (Text : Wide_Wide_String) return JSON_Value is
45     (null record);
46  function New_JSON_Object return JSON_Object is
47     (null record);
48  function New_JSON_Array return JSON_Array is
49     (null record);
50  function From_Object (Self : JSON_Object) return JSON_Value is
51     (null record);
52  function From_Array (Self : JSON_Array) return JSON_Value is
53     (null record);
54  end JSON;
55
56  function "+" (X : JSON.JSON_Object) return JSON.JSON_Value
57     renames JSON.From_Object;
58  function "-" (X : JSON.JSON_Array) return JSON.JSON_Value
59     renames JSON.From_Array;
60
61  Offices : JSON.JSON_Array :=
62     [+["name" => "North American Office",
63        "phones" => -[1_877_787_4628,
64                     1_866_787_4232,
65                     1_212_620_7300],
66        "email" => "info@adacore.com"],
67     +["name" => "European Office",
68        "phones" => -[33_1_49_70_67_16,
69                     33_1_49_70_05_52],
70        "email" => "info@adacore.com"]];
71  -----

```

(continues on next page)

(continued from previous page)

```
72 begin
73   -- Equivalent old initialization code is too long to print it here
74   null;
75 end Main;
```

### Code block metadata

Project: Courses.Ada\_2022\_Whats\_New.Container\_Aggregates\_4  
MD5: 3e8d96bbcf77e2c63fb87dcf313b98f1

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],
  "email" : "info@adacore.com"}]
```

## 6.1 References

- ARM 4.3.5 Container Aggregates<sup>21</sup>
- AI12-0212-1<sup>22</sup>

<sup>21</sup> <http://www.ada-auth.org/standards/22aarm/html/AA-4-3-5.html>

<sup>22</sup> <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0212-1.TXT>



## DELTA AGGREGATES

### 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 a 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:

```
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.

```
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

```
1  pragma Ada_2022;
2
3  with Ada.Text_IO;
4
5  procedure Main is
6
7      type Vector is record
8          X, Y, Z : Float;
9      end record;
10
11     Point_1 : constant Vector := (X => 1.0, Y => 2.0, Z => 3.0);
12     Projection_1 : constant Vector := (Point_1 with delta Z => 0.0);
13
14     type Vector_3D is array (1 .. 3) of Float;
15
16     Point_2 : constant Vector_3D := [1.0, 2.0, 3.0];
17     Projection_2 : constant Vector_3D := [Point_2 with delta 3 => 0.0];
18 begin
19     Ada.Text_IO.Put (Float'Image (Projection_1.X));
20     Ada.Text_IO.Put (Float'Image (Projection_1.Y));
21     Ada.Text_IO.Put (Float'Image (Projection_1.Z));
22     Ada.Text_IO.New_Line;
23     Ada.Text_IO.Put (Float'Image (Projection_2 (1)));
24     Ada.Text_IO.Put (Float'Image (Projection_2 (2)));
25     Ada.Text_IO.Put (Float'Image (Projection_2 (3)));
26     Ada.Text_IO.New_Line;
27 end Main;
```

## 7.3 References

- ARM 4.3.4 Delta Aggregates<sup>23</sup>
- AI12-0127-1<sup>24</sup>

---

<sup>23</sup> <http://www.ada-auth.org/standards/22aarm/html/AA-4-3-4.html>

<sup>24</sup> <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

```
1 pragma Ada_2022;  
2  
3 package Statistics is  
4  
5     type Statistic is record  
6         Count : Natural := 0;  
7         Total : Float := 0.0;  
8     end record;  
9  
10    My_Data : array (1 .. 5) of Float := [for J in 1 .. 5 => Float (J)];  
11  
12    Statistic_For_My_Data : Statistic;  
13  
14 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

```
1 pragma Ada_2022;  
2 with Ada.Text_IO;  
3  
4 procedure Main is  
5
```

(continues on next page)



(continued from previous page)

```
6  type Statistic is record
7      Count : Natural := 0;
8      Total : Float := 0.0;
9  end record;
10
11  My_Data : constant array (1 .. 5) of Float :=
12      [for J in 1 .. 5 => Float (J)];
13
14  Statistic_For_My_Data : Statistic;
15
16  begin
17      for Data of My_Data loop
18          Statistic_For_My_Data.Count := @ + 1;
19          Statistic_For_My_Data.Total := @ + Data;
20      end loop;
21
22      Ada.Text_IO.Put_Line (Statistic_For_My_Data'Image);
23  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

```
1  pragma Ada_2022;
2  with Ada.Text_IO;
3
4  procedure Main is
5
6      My_Data : array (1 .. 5) of Float := [for J in 1 .. 5 => Float (J)];
7
8      function To_Index (Value : Positive) return Positive is
9  begin
10         Ada.Text_IO.Put_Line ("To_Index is called.");
11         return Value;
12     end To_Index;
13
14  begin
15     My_Data (To_Index (1)) := @ ** 2 - 3.0 * @;
16     Ada.Text_IO.Put_Line (My_Data'Image);
17  end Main;
```

### Code block metadata

Project: Courses.Ada\_2022\_Whats\_New.Assignment\_Tagged\_3  
MD5: 98d6afb8aea5c0f6cd2bebe6b39962ad3

### 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!):

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

In Ada 2022, you can use a similar renaming:

```
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<sup>25</sup>](#)
- [AI12-0125-3<sup>26</sup>](#)

<sup>25</sup> <http://www.ada-auth.org/standards/22aarm/html/AA-5-2-1.html>

<sup>26</sup> <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0125-3.TXT>



## 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

```
1 with Ada.Text_IO;
2 with Ada.Integer_Text_IO;
3
4 procedure Main is
5 begin
6   Ada.Text_IO.Put ("Pos(False) =");
7   Ada.Integer_Text_IO.Put (Boolean'Pos (False));
8   Ada.Text_IO.New_Line;
9   Ada.Text_IO.Put ("Pos(True) =");
10  Ada.Integer_Text_IO.Put (Boolean'Pos (True));
11 end Main;
```

#### Code block metadata

Project: Courses.Ada\_2022\_Whats\_New.Enum\_Val.Pos  
MD5: de7c39f83f7df231dd648606579996a8

#### Runtime output

```
Pos(False) =      0
Pos(True)  =      1
```

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

Listing 2: main.adb

```
1 with Ada.Text_IO;
2
3 procedure Main is
4 begin
5     Ada.Text_IO.Put_Line (Boolean'Val (0)'Image);
6     Ada.Text_IO.Put_Line (Boolean'Val (1)'Image);
7 end Main;
```

### Code block metadata

Project: Courses.Ada\_2022\_Whats\_New.Enum\_Val.Val  
MD5: 43f712d25552970bccc4c0c84089d927

### Runtime output

```
FALSE
TRUE
```

## 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:

Listing 3: main.adb

```
1 with Ada.Text_IO;
2 with Ada.Integer_Text_IO;
3
4 procedure Main is
5     type My_Boolean is new Boolean;
6     for My_Boolean use (False => 3, True => 6);
7 begin
8     Ada.Text_IO.Put ("Enum_Rep(False) =");
9     Ada.Integer_Text_IO.Put (My_Boolean'Enum_Rep (False));
10    Ada.Text_IO.New_Line;
11    Ada.Text_IO.Put ("Enum_Rep(True)  =");
12    Ada.Integer_Text_IO.Put (My_Boolean'Enum_Rep (True));
13 end Main;
```

### Code block metadata

Project: Courses.Ada\_2022\_Whats\_New.Enum\_Val.Enum\_Rep  
MD5: 384ad9de7124c8131aa83ab71da58964

### Runtime output

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

And, for the reverse mapping, we can use **Type'Enum\_Val** (Int):

Listing 4: main.adb

```

1 with Ada.Text_IO;
2 with Ada.Integer_Text_IO;
3
4 procedure Main is
5   type My_Boolean is new Boolean;
6   for My_Boolean use (False => 3, True => 6);
7 begin
8   Ada.Text_IO.Put_Line (My_Boolean'Enum_Val (3)'Image);
9   Ada.Text_IO.Put_Line (My_Boolean'Enum_Val (6)'Image);
10
11   Ada.Text_IO.Put ("Pos(False) =");
12   Ada.Integer_Text_IO.Put (My_Boolean'Pos (False));
13   Ada.Text_IO.New_Line;
14   Ada.Text_IO.Put ("Pos(True) =");
15   Ada.Integer_Text_IO.Put (My_Boolean'Pos (True));
16 end Main;
```

**Code block metadata**

```
Project: Courses.Ada_2022_Whats_New.Enum_Val.Enum_Val
MD5: 6e06202472d4cf0ea7c68461ac7afcb1
```

**Runtime output**

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

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

```

1 with Ada.Text_IO;
2 with Ada.Integer_Text_IO;
3 with Ada.Unchecked_Conversion;
4
5 procedure Main is
6
7   type My_Boolean is new Boolean;
8   for My_Boolean use (False => 3, True => 6);
9   type My_Boolean_Int is range 3 .. 6;
10  for My_Boolean_Int'Size use My_Boolean'Size;
11
12  function To_Int is new Ada.Unchecked_Conversion
13    (My_Boolean, My_Boolean_Int);
14
15  function From_Int is new Ada.Unchecked_Conversion
```

(continues on next page)

(continued from previous page)

```
16     (My_Boolean_Int, My_Boolean);
17
18 begin
19     Ada.Text_IO.Put ("To_Int(False) =");
20     Ada.Integer_Text_IO.Put (Integer (To_Int (False)));
21     Ada.Text_IO.New_Line;
22     Ada.Text_IO.Put ("To_Int(True) =");
23     Ada.Integer_Text_IO.Put (Integer (To_Int (True)));
24     Ada.Text_IO.New_Line;
25     Ada.Text_IO.Put ("From_Int (3) =");
26     Ada.Text_IO.Put_Line (From_Int (3)'Image);
27     Ada.Text_IO.New_Line;
28     Ada.Text_IO.Put ("From_Int (6) =");
29     Ada.Text_IO.Put_Line (From_Int (6)'Image);
30 end Main;
```

### Code block metadata

Project: Courses.Ada\_2022\_Whats\_New.Enum\_Val.Conv  
MD5: 7c7624ed024b26036389f77dbd6cb109

### Runtime output

```
To_Int(False) =      3
To_Int(True)  =      6
From_Int (3)  =FALSE
From_Int (6)  =TRUE
```

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](#)<sup>27</sup>
- [AI12-0237-1](#)<sup>28</sup>

---

<sup>27</sup> <http://www.ada-auth.org/standards/22aarm/html/AA-13-4.html>

<sup>28</sup> <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0237-1.TXT>

## BIG NUMBERS

### **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));
```

```
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](https://en.wikipedia.org/wiki/RSA_(cryptosystem))<sup>29</sup> in a few lines of code. The main operation of RSA is  $(m^d) \bmod n$ . 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 `mod n` during the  $m^d$  calculation, it will work. Let's write this operation as a function:

---

<sup>29</sup> [https://en.wikipedia.org/wiki/RSA\\_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))



Listing 1: power\_mod.ads

```
1 pragma Ada_2022;
2
3 with Ada.Numerics.Big_Numbers.Big_Integers;
4 use Ada.Numerics.Big_Numbers.Big_Integers;
5
6 -- Calculate M ** D mod N
7
8 function Power_Mod (M, D, N : Big_Integer) return Big_Integer;
```

Listing 2: power\_mod.adb

```
1 function Power_Mod (M, D, N : Big_Integer) return Big_Integer is
2
3     function Is_Odd (X : Big_Integer) return Boolean is
4         (X mod 2 /= 0);
5
6     Result : Big_Integer := 1;
7     Exp    : Big_Integer := D;
8     Mult   : Big_Integer := M mod N;
9 begin
10    while Exp /= 0 loop
11        -- Loop invariant is Power_Mod'Result = Result * Mult**Exp mod N
12        if Is_Odd (Exp) then
13            Result := (Result * Mult) mod N;
14        end if;
15
16        Mult := Mult ** 2 mod N;
17        Exp := Exp / 2;
18    end loop;
19
20    return Result;
21 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](#)<sup>30</sup>. In that example, the *public key* is ( $n = 3233$ ,  $e = 17$ ) and the message is  $m = 65$ . The encrypted message is  $m^e \bmod n = 65^{17} \bmod 3233 = 2790 = c$ .

```
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  $c^d \bmod n = 2790^{413} \bmod 3233$ :

```
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:

---

<sup>30</sup> [https://en.wikipedia.org/wiki/RSA\\_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))

Listing 3: main.adb

```

1  pragma Ada_2022;
2
3  with Ada.Text_IO;
4  with Ada.Numerics.Big_Numbers.Big_Integers;
5  use  Ada.Numerics.Big_Numbers.Big_Integers;
6
7  procedure Main is
8
9      -- Calculate M ** D mod N
10
11     function Power_Mod (M, D, N : Big_Integer) return Big_Integer is
12
13         function Is_Odd (X : Big_Integer) return Boolean is
14             (X mod 2 /= 0);
15
16             Result : Big_Integer := 1;
17             Exp    : Big_Integer := D;
18             Mult   : Big_Integer := M mod N;
19         begin
20             while Exp /= 0 loop
21                 -- Loop invariant is Power_Mod'Result = Result * Mult**Exp mod N
22                 if Is_Odd (Exp) then
23                     Result := (Result * Mult) mod N;
24                 end if;
25
26                 Mult := Mult ** 2 mod N;
27                 Exp := Exp / 2;
28             end loop;
29
30             return Result;
31         end Power_Mod;
32
33     begin
34         Ada.Text_IO.Put_Line (Big_Integer'Image (2 ** 256));
35         -- Encrypt:
36         Ada.Text_IO.Put_Line (Power_Mod (M => 65, D => 17, N => 3233)'Image);
37         -- Decrypt:
38         Ada.Text_IO.Put_Line (Power_Mod (M => 2790, D => 413, N => 3233)'Image);
39     end Main;

```

**Code block metadata**

Project: Courses.Ada\_2022\_Whats\_New.Big\_Numbers\_Tiny\_RSA  
MD5: 6178da9d6998db6d51f31fd5c7cc5391

**Runtime output**

```

115792089237316195423570985008687907853269984665640564039457584007913129639936
2790
65

```

### 10.3 Big Reals

In addition to `Big_Integer`, Ada 2022 provides `Big Reals`<sup>31</sup>.

### 10.4 References

- [ARM A.5.6 Big Integers](#)<sup>32</sup>
- [ARM A.5.7 Big Reals](#)<sup>33</sup>
- [AI12-0208-1](#)<sup>34</sup>

---

<sup>31</sup> <http://www.ada-auth.org/standards/22aarm/html/AA-A-5-7.html>

<sup>32</sup> <http://www.ada-auth.org/standards/22aarm/html/AA-A-5-6.html>

<sup>33</sup> <http://www.ada-auth.org/standards/22aarm/html/AA-A-5-7.html>

<sup>34</sup> <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](#)<sup>35</sup> take a variable number of arguments and an ellipsis as the last parameter of the declaration. A typical and well-known example is:

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

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

```
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:

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

Unfortunately, doing it this way doesn't always work because some [ABI](#)<sup>36</sup>s use different calling conventions for variadic functions. For example, the [AMD64 ABI](#)<sup>37</sup> 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):

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

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

```
procedure printf_int  
(format : Interfaces.C.char_array;  
 value : Interfaces.C.int)
```

(continues on next page)

---

<sup>35</sup> <https://en.cppreference.com/w/c/variadic>

<sup>36</sup> [https://en.wikipedia.org/wiki/Application\\_binary\\_interface](https://en.wikipedia.org/wiki/Application_binary_interface)

<sup>37</sup> <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

(continued from previous page)

```
with Import,
  Convention   => C,
  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:

```
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:

```
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

```
1 with Interfaces.C;
2
3 procedure Main is
4
5   procedure printf_int
6     (format : Interfaces.C.char_array;
7      value  : Interfaces.C.int)
8   with Import,
9     Convention   => C_Variadic_1,
10    External_Name => "printf";
11
12 begin
13   printf_int (Interfaces.C.To_C ("d=%d"), 5);
14 end Main;
```

### Code block metadata

```
Project: Courses.Ada_2022_Whats_New.Variadic_Import
MD5: 94515f55a93f27e4f4ecec31256645d9
```

## 11.1 References

- ARM B.3 Interfacing with C and C++<sup>38</sup>
- AI12-0028-1<sup>39</sup>

---

<sup>38</sup> <http://www.ada-auth.org/standards/22aarm/html/AA-B-3.html>

<sup>39</sup> <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0028-1.TXT>