Ada Generic Averaging Function

Asked 9 years, 6 months ago Active 9 years, 6 months ago Viewed 2k times

4

I have a function which averages a certain numeric value from an array of records. This value is either a natural or an enumerated type delta. I have it summing up the values correctly but my question is this: how do I get the length of an array into a generic type, so that it can divide both integers and delta type numbers?

ada

edited Mar 7 '11 at 11:39

T.E.D.
40.3k 8 62 130

asked Mar 6 '11 at 21:12

n0dnarb
91 1 1 4

3 Answers

Active | Oldest | Votes

3

On your array-of-records use the 'Length attribute; this has the advantage of always working even if your bounds are somewhat odd, like -18..3, or an enumeration, like cheeses..fruits.

Something like:

```
Function Average( Input : In Array_of_Records ) Return float is
  -- You say you already have a summation function, so...
  Sum : Natural:= Summation( Input );
Begin
  Return Sum / Input'Length;
End Average;
```

You may need to convert the numeric types, by saying Float(Sum) or the like, as Ada does no automatic type "promotions."

edited Mar 6 '11 at 22:34

answered Mar 6 '11 at 21:17

Shark8
3,513 1 13 26

this only returns a float, i thought the poster asked for a function that would return a float or a discrete type? – NWS Mar 7 '11 at 17:33

True; the "return a float" portion of the 'or' is handled. – Shark8 Mar 7 '11 at 22:32

I interpreted that or as 'it will return this or that according to the instantiation of the generic', as the poster

"Digits <>", etc]... but there's probably good reason for that (compiler complexity, and that one might be tempted to use generic_number_1 & generic_number_2 with a generic function that tried to add the two together). – Shark8 Mar 12 '11 at 5:32

If you look at the actual type heirachy the 'real' types and 'integer' types are actually two separate subtrees bundled together as 'numeric'. i suspect this is why generics dont handle it too well. see en.wikibooks.org/wiki/Ada_Programming/Types#The_Type_Hierarchy – NWS Mar 14 '11 at 9:39

3

This has some flaws in it, but is this closer to what you wanted ?

NWS.

```
            end loop;
            return C;
         end Count;

         function Average (Vec : in Vec_T) return Element_T is
            S : constant Element_T := Sum (Vec);
            Len : constant Element_T := Count (Vec);
         begin
            return S / Len;
         end Average;
      end Arrayops;

      type Fl_Arr_T is array (Integer range <>) of Float;
      package Fl_Arr is new Arrayops (Element_T => Float,
                                      Zero => 0.0,
                                      One => 1.0,
                                      Vec_T => Fl_Arr_T);

      type Int_Arr_T is array (Integer range <>) of Integer;
      package Int_Arr is new Arrayops (Element_T => Integer,
                                       Zero => 0,
                                       One => 1,
                                       Vec_T => Int_Arr_T);


      My_Ints   : constant Int_Arr_T (1 .. 5) := (6,7,5,1,2);
      My_Floats : constant Fl_Arr_T (1 .. 7) := (6.1,7.2,5.3,1.4,2.5,8.7,9.7);

      Int_Sum   : constant Integer := Int_Arr.Sum (My_Ints);
      Int_Count : constant Integer := Int_Arr.Count (My_Ints);
      Int_Avg   : constant Integer := Int_Arr.Average (My_Ints);

      Float_Sum   : constant Float := Fl_Arr.Sum (My_Floats);
      Float_Count : constant Float := Fl_Arr.Count (My_Floats);
      Float_Avg   : constant Float := Fl_Arr.Average (My_Floats);

   begin

      Ada.Text_Io.Put_Line ("Integers => Sum: " & Integer'Image (Int_Sum) & ", Count: "
   & Integer'Image (Int_Count) & ", Avg: " & Integer'Image (Int_Avg));
```

Result :

edited Mar 7 '11 at 17:23

answered Mar 7 '11 at 17:10

NWS
2,875 1 14 32

1

Expanding on Shark8 a bit here...

Ada allows you to declare array types as unconstrained. Something like

```
type Array_of_Records is array (Natural range <>) of My_Record;
```

Gives you a type that can be used for arrays of records with starting and ending array indices that could be anywhere in the range of `Natural`.

One of the nifty things I can do with such a type is use it as a subroutine parameter, like so:

```
function Sum (Vector : in Array_of_Records) return Natural;
```

OK, so inside that routine, how do I know where the array bounds are? By using attributes, like so:

```
for index in Vector'first..Vector'last loop
```

or

```
for index in Vector'range loop
```

Of course for this to work, you must pass in a perfectly-sized array to your Sum routine. Supppose that isn't what you have. Suppose you instead have a huge array (kind of a buffer) and not all of the values are valid? Well, you keep track of what are the valid values, and pass in **only those** by using a *slice*.

```
Rec_Buffer : Array_of_Records (1..10_000);
Last_Valid_Rec : Natural := 0;
....
--// Rec_Buffer gets loaded with 2,128 values or something. We pass it into Sum
--// like so:
Ada.Text_IO ("Sum of vector is " &
             natural'image(Sum (Rec_Buffer (1..Last_Valid_Rec));
```

(warning - uncompiled code)

T.E.D.
40.3k 8 62 130