Implementing Constructor Calls with Parameters in ADA '95

by: Zeki Bayram
Boğaziçi University
Computer Engineering Dept.
Bebek 80815
Istanbul - Turkey
e-mail: bayram@boun.edu.tr

Abstract

Ada '95 does not provide for a constructor mechanism with parameters for initializing newly created objects. The package *Ada.Finalization* provides an **initialize** procedure which is automatically called at object creation time, but that procedure takes only one argument, i.e. the object being operated upon. This is hardly useful, since initialization of any kind usually requires additional information, which needs to be supplied through arguments. One way to get around the parameter problem is to specify a tagged type with discriminants that is derived from the **limited_controlled** type. The parameters that are meant to be given to the initialize procedure are then given as discriminants to the tagged record object. We describe this technique with a concrete example and explore its limitations and implications.

1. Introduction

Automatic object initialization once an object is created, and *cleanup* once an object is removed from memory are two important aspects of writing correct and maintainable object-oriented programs. Object oriented functionality has been added to the Ada language in the Ada '95 standard [[3]], but this does not include straight-forward support for automatically called initializer procedures (also called *constructors* in object oriented terminology) with parameters. Although the package *Ada.Finalization* provides two tagged type definitions (**controlled** and **limited_controlled**), both of which have an **initialize** procedure defined for them, and this **initialize** procedure is automatically called for any object of a type derived from either **controlled** or **limited_controlled** at creation time, only one parameter can be supplied to this procedure, i.e. the object being operated upon. Initialization, by its nature, requires values with which the object is to be initialized, either directly, or after some processing has been done on the values, and the inability of the **initialize** procedure to take more than one argument renders it practically useless for object initialization.

One way to get around the parameter problem can is by passing parameters as discriminants to the object. Unfortunately, not all types are permitted as parameters to tagged records with discriminants: only discrete types and access types. So in order to perform default initialization of objects with values from arbitrary types, access discriminants have to be used.

Another way might be seen as side-stepping the initialize procedure, and making an aggregate value assignment to the newly declared object. But this will not work if the object has a private part, which is generally the case in object-oriented programs.

In this article, we devise and explain a method of supplying parameters to the **initialize** procedure for types derived from the **limited_controlled** type of *Ada.Finalization*. In this method, pointers to actual parameters are passed as access discriminants to the tagged derived records. The initialize procedure can then use the values pointed to by the discriminants to perform any initialization action necessary. The actual parameters pointed to by the record discriminant are then disposed, since they are assumed not to be useful later on.

2. The package Ada.Finalization

Specification of the *Ada.Finalization* package which contains the procedure **finalize** is given in Figure 1. It is taken form the Ada Reference Manual [3].

```
package Ada.Finalization is
        pragma Preelaborate(Finalization);

        type Controlled is abstract tagged private;

        procedure Initialize(Object : in out Controlled);
        procedure Adjust              (Object : in out Controlled);
        procedure Finalize            (Object : in out Controlled);

        type Limited_Controlled is abstract tagged limited private;

        procedure Initialize(Object : in out Limited_Controlled);
        procedure Finalize  (Object : in out Limited_Controlled);
     private
        ... -- not specified by the language
     end Ada.Finalization;
```

Figure 1: Package Ada.Finalization

This package defines the types **controlled** and **limited_controlled.** Objects created using any type derived from **controlled** can be assigned and tested for equality. Hence there is a procedure **adjust** which is called as the final stage of an assignment operation, with the destination of the assignment being an argument. There is no **adjust** procedure for **limited_controlled** since by definition, objects created using any type derived from **limited_controlled** cannot be assigned or tested for equality.

All of the above procedures are called automatically by the compiler at various times: **initialize** is called right after an object is created (and as we have explained above, its usefulness is severely impaired by its inability to take extra arguments), **finalize** is called just before the space for an object is deallocated, either explicitly if the object was created dynamically, or implicitly, when the object is no longer in the scope of its declaration, and **adjust** is called as explained above.

3. Human-Man-Woman hierarchy and object initialization

We now explain and demonstrate the technique of passing extra arguments to the **initialize** procedure by using a concrete example, based upon the *human-man-woman* hierarchy of types presented in [[1]]. *human* is the base type, which defines common behavior for its two child classes, *man* and *woman*. Every human has a *name,* which is set at object creation time. *man* objects also have a *bearded* attribute, which can be set to a Boolean value at object creation time. Every human is greeted either as "Mr," "Mrs," or "XXXX" if no sex is specified.

In order for automatic initialization to take place, *man* is derived from **limited_controlled**. This choice is imposed upon us by the rules of ADA which prohibit any discriminants to records of types other than discrete or access. Since we want to be able to pass arguments of an arbitrary type to initialize an object, we have to pass them as access types. Since a type derived from **controlled** cannot have access discriminants, we are stuck with **limited_controlled** as the only choice.

In this example, we want to automatically initialize the *name* field of objects, hence we pass a string access value as a discriminant to the newly created object. For *man* objects, we can also specify whether he is bearded as a discriminant, and the discriminant need not be an access type since **Boolean** is a discrete type.

```
with text_io; use text_io;
with ada.finalization; use ada.finalization;

package Humanity is

subtype Name is String (1..30);
type Human(N: access string) is new limited_controlled with private;
procedure initialize(H: in out Human);
procedure finalize(H: in out Human);
procedure display(H: Human);
function Name_of(H: Human) return String;
function Title_of (H: Human) return String;
```

Figure 2: Specification of human

In Figure 2, we establish the environment to work in by importing definitions from **text_io** and **ada.finalization,** and give the primitive procedures (i.e. methods) for *human* objects. We note above that **N** is defined as an access discriminant to the type *Human*, which itself is a derived type of **limited_controlled**.

```
type Man(N:access string; B:boolean) is new Human(N) with private;
function Is_Bearded (M: Man) return Boolean;
procedure Shave (M: in out Man);
function Title_of (M: Man) return String;
procedure initialize(M: in out Man);

type Woman(N: access string) is new Human(N) with private;
function Title_of (W: Woman) return String;
```

Figure 3: Specification of man and woman

In Figure 3, we derive two new tagged types, *Man* and *Woman* from *Human.* We note that *Man* has two discriminants, **N** and **B**. **N** is passed directly to the parent type, *Man*. Both of these discriminants will be used by the procedure **initialize** to automatically initialize a *Man* object.

```
private
type Human(N:access string) is new limited_controlled with
  record
            First_Name: name;
            last_char: Natural := 0;
  end record;

type Man(N:access String; B:boolean) is new Human(N) with
  record
            Bearded: Boolean ;
  end record;

type Woman(N:access string) is new Human(N) with
  null record;

end Humanity;
```

Figure 4: Private parts of the declarations

Figure 4 contains the private parts of the declarations. In Figure 5, we start the implementation We need the **ada.unchecked_deallocation** generic procedure in order to define the **free** procedure for strings. The critical thing to notice is how the record discriminant **N** is used as a place holder for an extra argument to the **initialize** procedure. Also note that we cannot directly dispose of the object pointed to by the formal record parameter (discriminant) and that a local variable *s_a* is used to point to the object passed as an argument, and later disposed of.

```
with text_io; use text_io;
with ada;    use ada;
with ada.unchecked_deallocation;

package body Humanity is
type string_access is access all string;
procedure Free is new Ada.Unchecked_Deallocation (string, string_access);

procedure Display (H: Human) is
begin
          Put (Title_of (Human'Class (H)) & " " & Name_of(H) ); -- redispatching
          new_line;
end Display;

procedure initialize(H:in out Human) is
s_a: string_access := H.N;
begin
          put( "Human::initialize called for " & H.N.all);
          new_line;
          H.first_name(1.. H.N.all'last) := H.N.all;
          H.last_char := H.N.all'last;
          free(s_a);
 end initialize;

procedure finalize(H:in out Human) is
begin
  put("Human::finalize called for " & H.first_name(1..H.last_char));
  new_line;
end finalize;

function name_of(H: Human) return string is
begin
          return H.first_name(1..H.last_char);
end;

function title_of(H:Human) return string is
begin
          return "XXXX.";
end;
```

Figure 5: Implementation of Human operations

Next, we have the implementation of *man* and *woman* primitive functions/procedures. Since most of the behavior is inherited, not much work is needed here. Due to dynamic dispatching, the correct version of *title_of* will be called for each object. Also note that we defined an **initialize** procedure for *man* which first calls **man::initialize**, and then does its specific initialization.

```
function Is_Bearded (M: Man) return Boolean is
begin
        return M.bearded;
end;

procedure shave(M:in out man) is
begin
        M.bearded := false;
end;

function Title_of(M:man) return string is
begin
        return("Mr. ");
end;

procedure initialize(M:in out Man) is
begin
        initialize(human(M));
        put( "Man::initialize called for " & M.N.all);
        new_line;
        M.bearded := M.B;
end initialize;

function  title_of(W:woman) return string is
begin
        return "Mrs. ";
end;
end Humanity;
```

Figure 6: Implementation of man and woman operations

And finally, in Figure 7 we have the main procedure which imports the human-man-woman hierarchy and uses it, followed in Figure 8 by the program's output.

```
with Humanity; use Humanity;
with text_io; use text_io;

procedure main is

H: Human(new string'("any human"));
M: Man(new string'("joe"),true);
W: Woman(new string'("mary"));

begin
        display(H);
        display(M);
        display(W);
end main;
```

Figure 7: Main procedure

```
Human::initialize called for any human
Human::initialize called for joe
Man::initialize called for joe
Human::initialize called for mary
XXXX. any human
Mr.  joe
Mrs.  mary
Human::finalize called for mary
Human::finalize called for joe
Human::finalize called for any human
```

Figure 8: Program Output

4. Discussion

Below, we give some observations on the method described.

- Since objects in general cannot be passed as record discriminants directly (unless they are of a discrete or access type), access to the object must be passed in the case that the argument is not of a discrete type. Consequently, actual parameters (in the general case) must be created dynamically, and disposed of inside **initialize** for proper reclaiming of memory. Since access discriminants are read-only, a local variable must be made to point to the actual discriminant that will be disposed of.

- Since access discriminants are allowed only for limited types, types which need initialization should be derived from **limited_controlled**, rather than **controlled**. This is an annoying limitation: a **limited_controlled** object (or any object belonging to a type derived from **limited_controlled**) cannot be used as the target of an assignment operation, or tested for equality with another object of the same type.

- We note that we could have initialized the *bearded* attribute of a *man* object inside the record declaration, but we chose to do it this way to demonstrate how the **initialize** for *man* overrides the **initialize** for *human*, and that explicit type casting must be used to call the shadowed procedures.

5. Conclusion

Simply, we wish designers of the ADA '95 standard had thought of a better way than the package *Ada.Finalization* for implementing object initialization. We do not hesitate in suggesting a solution: slightly incrementing the syntax of the language to allow specifying constructor procedures. In the meantime, we will have to live with the limitations of the above described or any other work-around method if we wish to automatically initialize objects in ADA.

Acknowledgments: I would like to thank my colleague Levent Akın for his very useful comments and suggestions on an earlier draft of this article.

References

[1] Object-Oriented Programming with Ada 9X. HTML document by S. Barbey, M. Kempe, and A. Strohmeier.

[2] The Ada 95 Rationale, Intermetrics Inc. 1995

[3] The Ada Reference Manual, Intermetrics Inc. 1995