Using Finalization in Ada95

Mark Allen Weiss
School of Computer Science
Florida International University
University Park
Miami, FL 33199

Abstract

In Ada83, when a procedure (or function) terminates, all local variables "disappear." This process is called *finalization.* Generally, there is no problem, but occasionally unless additional work is performed, there will be a leak of resources. For example, if a file has been opened, we must remember to close it, and if memory has been allocated by a call to `new`, we may need to deallocate it by a call to `Unchecked_Deallocation`. Experience has shown that programmers are very forgetful in this regard. Ada95 provides a mechanism by which objects can specify the action that is to automatically occur during their finalization. We describe how this is done.

1 An Ada83 Stack Class

As a first example, consider the simple stack class described on pages 78 to 81. Here we use a linked list to maintain the stack items: the top of the stack is the first item in the list. Under this scenario, the `Stack` type is merely a pointer to a `Node`. Figure 1 shows the specification used in Ada83 (it reproduces Figure 3.41 in the text). The corresponding body is shown in Figure 2. In the original text, the procedure `Make_Null` was used to make an empty stack. However, this is clearly a memory leak: the statement `S:=null;` makes all the nodes in the former stack inaccessible, but still allocated. Thus the "new Ada83" implementation names this routine `Make_Empty`, and repeatedly calls `Pop` to delete items; `Pop` calls an instance of `Unchecked_Deallocation` and thus `Make_Empty` does not leak memory.

2 An Example of a Memory Leak

Consider now the following (admittedly silly) example in Figure 3.

Procedure `Inner` declares a local stack, and then pushes 100 items. The `Stack_List` class implements this by dynamically allocating a record for each `Push`. Under Ada83, when a call to `Inner` terminates, the object `Stack` exits scope, but the dynamically allocated memory remains allocated (and now unreferenced); thus Ada83 will leak memory. To fix this, the user would need to add a call to `Make_Empty`. This would plug the memory leak in our case (though it wouldn't if, for example, we used a linked list with a dynamically allocated header node to implement the stack). Few programmers will remember to free the memory, and moreover, it's not necessary if the static array implementation of the stack is used.

Although this simple example seems silly, it really isn't. For more complicated programs it is easy to find these kinds of situations and tracking down the memory leak can be very difficult, especially if the leak is slow. After all, if the program does not run out of memory, there will probably be little observable behavior, and it may take some time for a problem to manifest itself. Consequently memory leaks are pernicious problems.

```
generic
    type Element_Type is private;

package Stack_List is
    type Stack is limited private;

    procedure Make_Empty( S: in out Stack );
    procedure Push ( X: Element_Type; S: in out Stack );
    procedure Pop  ( S: in out Stack );
    function  Top  ( S: Stack ) return Element_Type;
    function  Is_Empty( S: Stack ) return Boolean;

    Underflow : exception;

private
    -- Stack implementation is a linked list with no header.

    type Node;
    type Stack is access Node;
    subtype Node_Ptr is Stack;

    type Node is
      record
        Element : Element_Type;
        Next    : Node_Ptr;
      end record;

end Stack_List;
```

Figure 1: Ada83 Stack_List specification

```
with Unchecked_Deallocation;

package body Stack_List is

    procedure Push( X: Element_Type; S: in out Stack ) is
    begin
        S := new Node'( X, S );
    end Push;

    procedure Dispose is new Unchecked_Deallocation( Node, Node_Ptr );

    procedure Pop( S: in out Stack ) is
        First_Cell : Node_Ptr;
    begin
        if Is_Empty( S ) then
            raise Underflow;
        end if;
        First_Cell := S;
        S := S.Next;
        Dispose( First_Cell );
    end Pop;

    function Is_Empty( S: Stack ) return Boolean is
    begin
        return S = null;
    end Is_Empty;

    procedure Make_Empty( S: in out Stack ) is
    begin
        while not Is_Empty( S ) loop
            Pop( S );
        end loop;
    end Make_Empty;

    function Top( S: Stack ) return Element_Type is
    begin
        if Is_Empty( S ) then
            raise Underflow;
        end if;
        return S.Element;
    end Top;

end Stack_List;
```

Figure 2: Ada83 Stack_List body

```
with Stack_List;

procedure Outer is
    package My_Stack is new Stack_List( Integer ); use My_Stack;

    -- Push 100 items onto the stack.
    -- With linked list, this makes 10 calls to new.
    -- This is a memory leak in Ada83.
    procedure Inner is
        S : Stack;
    begin
        for i in 1..100 loop
            Push( i, S );
        end loop;
        -- Need Make_Empty( S ); here
    end Inner;
begin
    for i in 0..100_000 loop
        Inner;
    end loop;
end Outer;
```

Figure 3: Example of a memory leak in Ada83 that can be fixed in Ada95

One of the problems is that if programmers (i.e. users of the `Stack`) have to manage memory themselves, they may do it wrong. The `Stack` ought to be able to specify what happens to memory it has allocated when instances of the `Stack` exit scope.

3 The package `Ada.Finalization`

The package `Ada.Finalization` defines two types, namely `Controlled` and `Limited_Controlled`, the difference being that the latter is a limited type. Perhaps the most important procedure that is defined for these types is

```
procedure Finalize( Object : in out Controlled );
```

Procedure `Finalize` is called whenever an object leaves scope (for instance on local objects immediately prior to a return statement, exception, or other exit of the block in which it is declared). `Finalize` has a null body and does nothing. The general idea then is that we can make `Stack` a derived type of `Limited_Controlled`, and then redefine procedure `Finalize` for the `Stack`. In other words, we would provide, in class `Stack_List` the new dispatching operation

```
procedure Finalize( S : in out Stack );
```

This operation would deallocate any memory that was associated with the stack `S` (presumably by calling `Make_Empty`). The user of the stack would be freed from this burden.

4 An Ada95 Stack Class

```
with Ada.Finalization;

generic
    type Element_Type is private;

package Stack_List is
    type Stack is new Ada.Finalization.Limited_Controlled with private;

    procedure Finalize( S: in out Stack );

    procedure Make_Empty( S: in out Stack );
    procedure Push ( X: Element_Type; S: in out Stack );
    procedure Pop  ( S: in out Stack );
    function  Top  ( S: Stack ) return Element_Type;
    function  Is_Empty( S: Stack ) return Boolean;

    Underflow: exception;

private
    -- Stack implementation is a linked list with no header.

    type Node;
    type Node_Ptr is access Node;

    type Node is
      record
        Element : Element_Type;
        Next    : Node_Ptr;
      end record;

    type Stack is new Ada.Finalization.Limited_Controlled with
      record
        TopOfStack : Node_Ptr;
      end record;

end Stack_List;
```

Figure 4: Ada95 Stack_List specification

The Ada95 version of Stack_List requires a few changes. First, as mentioned above, the Stack is now a derived type, rather than merely a pointer. Consequently, in the visible part of the specification, we say that Stack is new Limited_Controlled with additional data members specified in the private section. The additional member, as we can see, is the field TopOfStack of type Node_Ptr. Notice that now Node_Ptr must be defined as an access type directly (rather than its former definition of being a subtype of Stack. We also provide the declaration for Finalize. This is shown in Figure 4.

The new body is shown in Figure 5. Most of the changes involve the fact that in the old implementation the pointer to the header node is S, whereas now it is S.TopOfStack. The body of Finalize must also be provided: We see that it is implemented as a simple call to Make_Empty.

5 Initialize

The Controlled and Limited_Controlled types also allow the definition of an initialization procedure that is performed when an object is created. The prototype is

```
procedure Initialize( Object : in out Controlled );
```

Because Ada83 has already provided that members of a record can be initialized (and that pointers are initialized to null by default), most objects do not require special initialization action. For instance, the Stack is automatically created with TopOfStack correctly initialized, and this is true for both the current list implementation and the array implementation in the text.

As a more complicated example, suppose we want to declare a List. The implementation in the text uses a header dummy node, but it requires that the user call Make_Empty to allocate this node. A more reliable mechanism is to have the procedure Initialize call Make_Empty. We would also want Finalize to first call Make_Empty to remove the items in the list, and then have it call dispose to remove the dummy header node.

Figure 6 shows part of the new Linked_List specification. The changes are very similar to those made for the Stack_List class. We indicate that both Initialize and Finalize will be supplied. The most interesting thing that emerges here is a clear distinction between the type Position and List; in the Ada83 version of the code, they were different types by virtue of the declaration, but they looked the same. Here they don't even look the same.

Three of the routines that are in the body are shown in Figure 7. Make_Empty is rewritten to allocate the header node if none exists. If a header node is already present, the list is made empty by a call to Delete_List. Initialize and Finalize are simple routines that directly implement the description above.

6 Adjust

Suppose we have two lists L1 and L2 and we want to allow

```
L1 := L2;      -- Copy two lists
```

As it stands right now, this is illegal because the list is a limited private type. The reason the list is limited private is that the meaning of := is a copy of the data members, namely the pointers Header. If List was a private type, then the assignment above would merely have L1 and L2 share a pointer to the header node; we would not have the independent versions that is typical of the copy. C++ considers the copy assignment to be an operator, and thus its meaning can be altered by the user. Ada95 allows a way to make copying meaningful for private types.

The mechanism is procedure Adjust. After the default copy is performed, procedure Adjust is called. In the case of linked lists, we could have Adjust replicate a copy of the linked list. The new specification is shown in Figure 8. Then when Adjust is called, L.Header represents a pointer to the list header that would otherwise be shared by two lists. To avoid this, we make a copy of the list as shown in Figure 9.

```
with Unchecked_Deallocation;

package body Stack_List is

    procedure Push( X: Element_Type; S: in out Stack ) is
    begin
        S.TopOfStack := new Node'( X, S.TopOfStack );
    end Push;

    procedure Dispose is new Unchecked_Deallocation( Node, Node_Ptr );

    procedure Pop( S: in out Stack ) is
        First_Cell : Node_Ptr;
    begin
        if Is_Empty( S ) then
            raise Underflow;
        end if;
        First_Cell := S.TopOfStack;
        S.TopOfStack := S.TopOfStack.Next;
        Dispose( First_Cell );
    end Pop;

    function Is_Empty( S: Stack ) return Boolean is
    begin
        return S.TopOfStack = null;
    end Is_Empty;

    procedure Make_Empty( S: in out Stack ) is
    begin
        while not Is_Empty( S ) loop
            Pop( S );
        end loop;
    end Make_Empty;

    function Top( S: Stack ) return Element_Type is
    begin
        if Is_Empty( S ) then
            raise Underflow;
        end if;
        return S.TopOfStack.Element;
    end Top;

    procedure Finalize( S: in out Stack ) is
    begin
        Make_Empty( S );
    end Finalize;

end Stack_List;
```

Figure 5: Ada95 Stack_List

```
with Ada.Finalization;

generic
    type Element_Type is private;
    with function "="( Left, Right: Element_Type ) return Boolean;

package Linked_Lists is
    type Position is private;
    type List is new Ada.Finalization.Limited_Controlled with private;

    procedure Initialize( L: in out List );
    procedure Finalize( L: in out List );

    -- Remaining dispatch operations are omitted for compactness

private
    type Node is
      record
        Element : Element_Type;
        Next    : Position;
      end record;

    type Position is access Node;

    type List is new Ada.Finalization.Limited_Controlled with
      record
        Header : Position;
      end record;

end Linked_Lists;
```

Figure 6: Part of the Ada95 Linked_Lists specification

Is this such a great idea? Probably not! Experience with C++ has shown that in many cases it is better to use a limited private type rather than attempting to provide a new meaning, because of the hidden costs of providing these copies (and the fact that copies may be made unintentionally).

The alert reader may wonder about a potential problem: in the statement `L1:=L2;` what happens if `L1` was a non-empty list? After all, by clobbering `L1.Header`, haven't we leaked memory? The answer is that prior to assignment, finalization is always performed.

7 Debugging Tip

Your program works, gives all the answers, and then either crashes or hangs in an infinite loop after the last statement is executed. What is wrong?

Almost certainly, something has gone wrong with a `Finalize` procedure. Put some printing statements there to see what the problem is.

8 Exercises

1. Give a linked list-based implementation of the queue that includes a `Finalize` procedure.

2. Implement the `Cursor_Lists` package. This will involve both `Initialize` and `Finalize`.

```
with Unchecked_Deallocation;    -- Place this before the package body.
package body Linked_Lists is

    -- Most of the dispatch routines are omitted for compactness.

    procedure Make_Empty( L: in out List ) is
    begin
        if L.Header = null then
            L.Header := new Node;
            L.Header.Next := null;
        else
            Delete_List( L );
        end if;
    end Make_Empty;

    procedure Initialize( L: in out List ) is
    begin
        Make_Empty( L );
    end Initialize;

    procedure Finalize( L: in out List ) is
    begin
        Delete_List( L );
        Dispose( L.Header );
    end Finalize;

end Linked_Lists;
```

Figure 7: Part of the Ada95 Linked_Lists body

```
with Ada.Finalization;

generic
    type Element_Type is private;
    with function "="( Left, Right: Element_Type ) return Boolean;
package Linked_Lists is
    type Position is private;
    type List is new Ada.Finalization.Controlled with private;

    procedure Initialize( L: in out List );
    procedure Adjust( L: in out List );
    procedure Finalize( L: in out List );

    -- Remaining dispatch operations are omitted for compactness
private
    type Node is
      record
        Element : Element_Type;
        Next    : Position;
      end record;

    type Position is access Node;

    type List is new Ada.Finalization.Controlled with
      record
        Header : Position;
      end record;
end Linked_Lists;
```

Figure 8: Making `List` private: I don't care for this myself

```
    procedure Adjust( L: in out List ) is
        OldPosition : Position := L.Header.Next;
        NewPosition : Position;
    begin
        L.Header.Next := null;
        NewPosition := L.Header;
        while OldPosition /= null loop
            NewPosition.Next := new Node'( OldPosition.Element, null );
            OldPosition := OldPosition.Next;
            NewPosition := NewPosition.Next;
        end loop;
    end Initialize;
```

Figure 9: `Adjust` for private `Lists`