The following contributions are taken from the AdaCore “Gem of the Week” series. Gems are expert tips and insights that will help you get the most out of Ada 2005. The full collection of gems can be found at http://www.adacore.com/home/ada_answers, where you may obtain the source code and post comments as well.

Gem #10: Limited Types in Ada 2005 — Extended Return Statements

Author: Bob Duff, AdaCore

Abstract: Ada Gem #10 — An extended_return_statement may be used to give a name to the result object being created by a function.

Let’s get started…

A common idiom in Ada 95 is to build up a function result in a local object, and then return that object:

```
function Sum (A : Array_Of_Natural) return Natural is
   Result : Natural := 0;
begin
   for Index in A’Range loop
      Result := Result + A (Index);
   end loop;
   return Result;
end Sum;
```

Ada 2005 allows a notation called the extended_return_statement, which allows you to declare the result object and return it as part of one statement. It looks like this:

```
function Sum (A : Array_Of_Natural) return Natural is
begin
   return Result : Natural := 0 do
      for Index in A’Range loop
         Result := Result + A (Index);
      end loop;
   end return;
end Sum;
```

The return statement here creates Result, initializes it to 0, and executes the code between “do” and “end return”. When “end return” is reached, Result is automatically returned as the function result.

For most types, this is no big deal — it’s just syntactic sugar. But for limited types, this syntax is almost essential:

```
function Make_Task (Val : Integer) return Task_Type is
   Result : Task_Type (Discriminant => Val * 3);
begin
   … -- some statements
   return Result; -- Illegal!
   end Make_Task;
```

The return statement here is illegal, because Result is local to Make_Task, and returning it would involve a copy, which makes no sense (which is why task types are limited). In Ada 2005, we can write constructor functions for task types:

```
   function Make_Task (Val : Integer) return Task_Type is
   begin
      return Result : Task_Type (Discriminant => Val * 3) do
         … -- some statements
      end return;
   end Make_Task;
```

If we call it like this:

```
    My_Task : Task_Type := Make_Task (Val => 42);
```

Result is created "in place" in My_Task. Result is temporarily considered local to Make_Task during the "… -- some statements" part, but as soon as Make_Task returns, the task becomes more global. Result and My_Task really are one and the same object.

When returning a task from a function, it is activated after the function returns. The "… -- some statements" part had better not try to call one of the task's entries, because that would deadlock. That is, the entry call would wait until the task reaches an accept statement, which will never happen, because the task will never be activated.

While the extended_return_statement was added to the language specifically to support limited constructor functions, it comes in handy whenever you want a local name for the function result:

```
   function Make_String (…) return String is
      Length : Natural := 10;
   begin
      if … then
         Length := 12;
      end if;
      return Result : String (1..Length) do
         … -- fill in the characters
         pragma Assert (Is_Good (Result)); null;
      end return;
   end Make_String;
```

Gem #11: Limited Types in Ada 2005 — Constructor Functions Part 2

Author: Bob Duff, AdaCore

Abstract: Ada Gem #11 — We show here how limited constructor functions can be used in various contexts to build new limited objects in place.

Let's get started…

We've earlier seen examples of constructor functions for limited types similar to this:

```
package P is
   type T (<>) is limited private;
   function Make_T (Name : String) return T; -- constructor function
private
   type T is new Limited_Controlled with
      record
         …
      end record;
end P;

package body P is
   function Make_T (Name : String) return T is
   begin
      return (Name => To_Unbounded_String (Name), others => <>);
   end Make_T;
end P;

function Make_Rumplestiltskin return T is
begin
    return Make_T (Name => "Rumplestiltskin");
end Make_Rumplestiltskin;
```

It is useful to consider the various contexts in which these functions may be called. We've already seen things like:

```
Rumplestiltskin_Is_My_Name : T := Make_Rumplestiltskin;
```

in which case the limited object is built directly in a standalone object. This object will be finalized whenever the surrounding scope is left.

We can also do:

```
procedure Do_Something (X : T);

Do_Something (X => Make_Rumplestiltskin);
```

Here, the result of the function is built directly in the formal parameter X of Do_Something. X will be finalized as soon as we return from Do_Something.

We can allocate initialized objects on the heap:

```
type T_Ref is access all T;
Global : T_Ref;

procedure Heap_Alloc is
   Local : T_Ref;
begin
   Local := new T'(Make_Rumplestiltskin);
   if … then
      Global := Local;
   end if;
end Heap_Alloc;
```

The result of the function is built directly in the heap-allocated object, which will be finalized when the scope of T_Ref is left (long after Heap_Alloc returns).

We can create another limited type with a component of type T, and use an aggregate:

```
type Outer_Type is limited
   record
      This : T;
      That : T;
   end record;

Outer_Obj : Outer_Type := (This => Make_Rumplestiltskin,
                           That => Make_T (Name => “”));
```

As usual, the function results are built in place, directly in Outer_Obj.This and Outer_Obj.That, with no copying involved.

The one case where we _cannot_ call such constructor functions is in an assignment statement:

```
Rumplestiltskin_Is_My_Name := Make_T(Name => ""); -- Illegal!
```

which is illegal because assignment statements involve copying. Likewise, we can’t copy a limited object into some other object:

```
Other : T := Rumplestiltskin_Is_My_Name; -- Illegal!
```

Gem #12: Limited Types in Ada 2005 — <> Notation Part 2

Author: Bob Duff, AdaCore

Abstract: Ada Gem #12 — We show how the <> notation in aggregates may be used to make better use of record-component default values, thus avoiding duplication of code.

Let's get started…

Have you ever written Ada 95 code like this?

```
package P is
   type T is private;
   …
private
   type T is
      record
         Color : Color_Enum := Red;
         Is_Gnarly : Boolean := False;
         Count : Natural;
      end record;
end P;

package body P is
   Object_100 : constant T :=
      (Color => Red, Is_Gnarly => False, Count => 100);
   …
end P;
```

We want Object_100 to be a default-initialized T, with Count equal to 100. It's a little bit annoying that we had to write the default values Red and False twice. What if we change our mind about Red, and forget to change it in all the relevant places?

The "<>" notation comes to the rescue. If we want to say, "make Count equal 100, but initialize Color and Is_Gnarly to their defaults", we can do this:

```
Object_100 : constant T :=
   (Color => <>, Is_Gnarly => <>, Count => 100);
```

On the other hand, if we want to say, "make Count equal 100, but initialize all other components, including the ones we might add next week, to their defaults", we can do this:

```
Object_100 : constant T := (Count => 100, others => <>);
```

Note that if we add a component "Glorp : Integer;" to type T, then the "others" case leaves Glorp undefined just as this Ada 95 code would do:

```
Object_100 : T;
```

```
   Object_100.Count := 100;
```

Think twice before using “others”.

Gem #13: Interrupt Handling Idioms (Part 1)

Author: Pat Rogers, AdaCore

Abstract: Ada Gem #13 — There are two design idioms commonly used when handling interrupts with Ada. One has more of the characteristics associated with good software engineering, the other somewhat better performance. We explore these two idioms in this gem.

Let's get started…

Recall that, in Ada, protected procedures are the standard interrupt-handling mechanism. This approach has a number of advantages over the "traditional" use of non-protected procedures. First, normal procedures don't have a priority, but protected objects can have an interrupt priority assigned and are thus integrated with the overall priority semantics. Execution of entries and procedures within the protected object will execute at that level and only higher-level interrupts can preempt that execution. Thus no race conditions are possible either. Additionally, condition synchronization is expressed directly, via entry barriers, making interaction with other parts of the system easy to express and understand. Finally, protected objects support localization of data and their manipulating routines, as well as localization of multiple interrupt handlers within one protected object when they each need access to the local data.

The response to interrupts is often arranged in levels, with a first-level handler providing a very fast response that does limited processing and a secondary-level handler that does more expensive processing outside of the interrupt context, at application-level priority. A natural expression of this structure is to use a protected procedure as the first-level handler and a task as the secondary level. The protected procedure responds to the interrupt and then signals the task when it should run.

For example, consider message handling over a UART (Universal Asynchronous Receiver Transmitter), in which an interrupt signals arrival of the first character. The interrupt handler procedure would capture that character, place it into a buffer within the protected object, and then either poll for the remaining characters (if appropriate) or reset for the next interrupt. Once the entire message is received the protected procedure could then signal the secondary handler task to parse the message and respond accordingly.

We will implement such a message processor using both design idioms. In each case we encapsulate both levels of the interrupt handling code inside the body of a package named `Message_Processor`. All the processing is done in the package body and nothing is exported that requires a completion. Hence we need a pragma `Elaborate_Body` in the declaration to make the package body legal. The content of the package declaration, as shown below, would probably go in the body as well, but for the purpose of this gem we will leave them here.

```
package Message_Processor is

  pragma Elaborate_Body;

  subtype Message_Size is Integer range 1 .. 256; — arbitrary

  type Contents is array (Message_Size range <>) of Character;

  type Message (Size : Message_Size) is
    record
      Value  : Contents (1..Size);
      Length : Natural := 0;
    end record;

end Message_Processor;
```

First Design Idiom

In the first idiom the first-level protected procedure handler signals the second-level task handler by enabling a barrier on a protected entry in the same protected object. The second-level task suspends on the entry call and, when allowed to resume execution, performs the secondary processing. Entry parameters can be used to pass information to the task, for example the message received on the UART. The code for this idiom results in a package body structured as follows:

```
with UART;
with System;
with Ada.Interrupts;

package body Message_Processor is

   Port : UART.Device;

   protected Receiver is … — the first-level handler

   protected body Receiver is …

   task Process_Messages is …

   task body Process_Messages is …

begin
   UART.Configure (Port, UART_Data_Arrival, UART_Priority);
   UART.Enable_Interrupts (Port);
end Message_Processor;
```

In the above, the protected object `Receiver` is the first-level handler; the secondary handler is the task `Process_Messages`. The UART hardware is represented by the object named `Port`, of a type defined by package `UART` (not shown). The package body executable part automatically configures the UART and enables its interrupts after the protected object and task are elaborated.

The protected object `Receiver` contains the interrupt-handling procedure, the entry to be called by the secondary handler task, a buffer containing the currently received characters, and a boolean variable used for the entry barrier:

```
UART_Priority     : constant System.Interrupt_Priority   := …
UART_Data_Arrival : constant Ada.Interrupts.Interrupt_Id := …

protected Receiver is
   entry Wait (Msg : access Message);
   pragma Interrupt_Priority (UART_Priority);
private
   procedure Handle_Incoming_Data;
   pragma Attach_Handler (Handle_Incoming_Data, UART_Data_Arrival);
   Buffer        : Contents (Message_Size);
   Length        : Natural := 1;
   Message_Ready : Boolean := False;
end Receiver;
```

The pragma `Interrupt_Priority` assigns the given priority to the whole protected object. No other interrupts at or below that level will be enabled whenever the procedure is executing. Note that the procedure is declared in the private part of the protected object. Placement there precludes “accidental” calls from client software in future maintenance activities. Note also the pragma `Attach_Handler` that permanently ties the procedure to the interrupt.

In the body of the protected object we have the bodies for the entry and the procedure. The entry is controlled by the boolean `Message_Ready` that is set to `True` when the interrupt handler determines that all the characters have been received for a given message. The entry body copies the buffer content directly into the caller’s Message object and then resets the buffer for the next message arrival.

```
protected body Receiver is

   entry Wait (Msg : access Message) when Message_Ready is
   begin
      Msg.Value (1 .. Length) := Buffer (1 .. Length);
      Msg.Length := Length;
      — reset for next arrival
      Length := 1;
      Message_Ready := False;
   end Wait;

   procedure Handle_Incoming_Data is
   begin
      UART.Disable_Interrupts (Port);
      Buffer (1) := UART.Data (Port);
      — poll for all remaining
      while UART.Data_Available (Port) loop
         Length := Length + 1;
         Buffer (Length) := UART.Data (Port);
      end loop;
      UART.Enable_Interrupts (Port);
      — wake up the task
```

```
         Message_Ready := True;
      end Handle_Incoming_Data;

   end Receiver;
```

The interrupt handler procedure uses the polling approach in this example. It first disables further interrupts from the UART and then captures all the incoming characters. Finally, in re-enables the device interrupts and enables the entry by setting Message Ready to `True`.

The second-level handler task has no entries of its own because nothing calls it. We only need to set the priority of the task, as specified in package `Config` (not shown) that defines all the priorities of the application.

```
   task Process_Messages is
      pragma Priority (Config.Process_Messages_Priority);
   end Process_Messages;

   task body Process_Messages is
      Next_Message : aliased Message (Size => Message_Size'Last);
   begin
      — any initialization code
      loop
         Receiver.Wait (Next_Message'Access);
         — process Next_Message …
      end loop;
   end Process_Messages;
```

The task suspends until the entry is executed and then processes the message is some application-defined way.

Next week we will explore the second design idiom and then compare the two. Stay tuned for more…

Gem #14: Interrupt Handling Idioms (Part 2)

Author: Pat Rogers, AdaCore

Abstract: Ada Gem #14 — There are two design idioms commonly used when handling interrupts with Ada. One has more of the characteristics associated with good software engineering, the other somewhat better performance. We explore these two idioms in this gem.

Let's get started…

Last week we introduced this topic and explored the first of the two designs. The gist of it is that we want to implement a multi-level response to interrupts, in which a protected procedure implements the first-level handler and a task implements the secondary level handling outside of the interrupt context. We use serial communications over a UART (Universal Asynchronous Receiver Transmitter) as the example. In both designs we encapsulate the two levels of the interrupt handling code inside the body of a package named `Message_Processor`.

Second Design Idiom

The second idiom is similar to the first, in that a protected object encapsulates the interrupt handling procedure, but this second design uses a `Suspension_Object`, instead of a protected entry, to signal the task. The type `Suspension_Object` is declared within package `Ada.Synchronous_Task_Control` and is essentially a boolean flag with synchronization. A single task can await a given `Suspension_Object` becoming "true" and will suspend until another task sets that state. The resulting structure for the package body is as follows:

```
with UART;
with System;
with Ada.Interrupts;
with Ada.Synchronous_Task_Control;

package body Message_Processor is

  Port : UART.Device;

  package STC renames Ada.Synchronous_Task_Control;

  Message_Ready : STC.Suspension_Object;

  Buffer : Contents (Message_Size);
  Length : Natural := 1;

  protected Receiver is …

  protected body Receiver is …

begin
```

```
  UART.Configure (Port, UART_Data_Arrival, UART_Priority);
  UART.Enable_Interrupts (Port);
end Message_Processor;
```

The significant differences in structure include the new `Suspension_Object` and the buffer containing the most recently received message, moved from within the protected object to the package body declarative part.

The protected object Receiver no longer declares an entry because it uses `Message_Ready` to signal the task. No local data are required either. Only the interrupt handling procedure is required.

```
  protected Receiver is
    pragma Interrupt_Priority (UART_Priority);
  private
    procedure Handle_Incoming_Data;
    pragma Attach_Handler (Handle_Incoming_Data, UART_Data_Arrival);
  end Receiver;
```

The body of the procedure is identical to that of the other design idiom except that it sets the `Suspension_Object` to `True` instead of setting a local boolean variable used in an entry barrier.

```
  protected body Receiver is

    procedure Handle_Incoming_Data is
    begin
      — as before
      …

      STC.Set_True (Message_Ready);
    end Handle_Incoming_Data;

  end Receiver;
```

Now the task does the work that the entry did in the other design. It first waits for `Message_Ready` to be `True`, suspending if necessary. Then it copies the buffer content to the local Message object and resets for the next arrival. Note that when `Suspend_Until_True` executes, the specified `Suspension_Object` is automatically set to `False` on exit.

```
  task body Process_Messages is
    Next_Message : aliased Message (Size => Message_Size’Last);
  begin
    — any initialization code here…
    loop
      STC.Suspend_Until_True (Message_Ready);
      — capture the new message in Buffer
      Next_Message.Value (1..Length) := Buffer (1..Length);
      Next_Message.Length  := Length;
      — reset for next arrival
      Length := 1;
      — process Next_Message …
    end loop;
  end Process_Messages;
```

Design Idiom Comparison

The first idiom, in which an entry is used to signal the second-level handler, is a better design from a software engineering point of view: functionality is grouped with the manipulated data (the buffer is local to the protected object), condition synchronization is directly expressed via the entry barrier and is not implemented by the calling task, and communication is accomplished via the entry parameters. All these characteristics result in a more maintainable, robust implementation.

However, the semantics of protected objects implies a run-time cost that, albeit relatively small, is greater than that of a `Suspension_Object`. The second idiom is very likely to be faster than the first, assuming a decent implementation of the type `Suspension_Object`. However, placing all the code within the package body limits the deleterious effects of the resulting structure.

The interrupt handling and management facilities provided by Ada are defined in the Systems Programming Annex, section C.3, of the Reference Manual. Support is extensive and is well worth studying.

Gem #15: Timers

Author: Anh Vo, Santa Clara, California

Abstract: Ada Gem #15 — Timers are essential software elements of embedded and real-time systems. Thus, making an intuitive and easy way to create timers helps in the software design process.

Let’s get started…

Timers are essential software elements of embedded and real-time systems. Thus, making an intuitive and easy way to create timers helps in the software design process. Ada provides a predefined package named Ada.Real_Time.Timing_Events for doing just that. With this package, a timer, one-shot or periodic one, can be created with a breeze. Furthermore, these timers can be genericized for different durations. Going one step further these generic timers can be combined in a single generic package as shown below along with the test codes. Note that the test codes are pretty crude. However, the output is printed out for each second approximately. Thus, the test results can be checked using one thousand count rule, one thousand one, one thousand two…

Here is an example for creating a 250 millisecond one-shot timer.

```
with Generic_Timers;
with Ada.Real_Time;
with Ada.Text_Io;
…
declare
   use Ada;
   Span : constant Time_Span := Real_Time.Milliseconds (250);
   Timer_Id : constant String := “250 Millisecond One Shot timer”;

   procedure Timer_Handler is
   begin
      Put_Line (”Do whatever is necessary when the timer expires”);
   end Timer_Handler;

   package The_Timer is new Generic_Timers (True, Timer_Id, Span,
Timer_Hander); –timer object
begin
   …
   The_Timer.Start;
   …
end;
…
```

Other one-shot timers and periodic timers for different durations are in the test package Timers_Test. Note that at my current project creating a timer takes five to ten times longer and much less intuitive. On this note, enjoy it with a smile.

Gem #16: Pragma No_Return

Author: Bob Duff, AdaCore

Abstract: Ada Gem #16 — **Pragma** No_Return may be used to mark procedures that cannot return normally.

Let's get started…

Did he ever return,
No he never returned
And his fate is still unlearn'd
He may ride forever
'neath the streets of Boston
He's the man who never returned.
– from "Charlie on the M.T.A."
– by The Kingston Trio

It is occasionally useful to have a procedure that never returns normally, where "normally" means reaching the "end" or executing a "return;". For example, a procedure might format an error message based on its parameters, send the message to a log file, and then unconditionally raise an exception. Other ways to avoid returning normally are to loop forever, and to wait upon an entry barrier that never becomes True.

Such procedures are unusual, and therefore deserve to be documented. A comment works, but **pragma** No_Return is better because the compiler makes sure that the procedure does not, in fact, return:

```
   procedure Log_Error (…);
   pragma No_Return (Log_Error);  –  Mark it as a non-returning
procedure.

   procedure Log_Error (…) is
   begin
      Put_Line (…);
      raise Some_Error;
   end Log_Error;
```

Pragma No_Return in Ada 2005 was inspired by the implementation-defined **pragma** No_Return that has existed in GNAT for some time. Now it's a standard feature of the Ada language.

According to the Ada RM, if a non-returning procedure reaches the "end", Program_Error is raised. But that's just a tripping hazard; what we really want is a compile-time check. In fact, if there is any control-flow path that can reach the "end" of a supposedly non-returning procedure, GNAT will give a warning. As always, you can tell GNAT to treat warnings as errors. These warnings are necessarily conservative, so you might sometimes need to use **`pragma`** `Warnings (Off)` to prevent the compiler from "crying wolf".

See also paragraph 6.5.1(1.a/2) of the Annotated Ada Reference Manual, regarding **`pragma`** `No_Deposit`: http://www.adaic.org/standards/05aarm/html/AA-6-5-1.html

Gem #17: Pragma No_Return, Part 2 (functions)

Author: Bob Duff, AdaCore

Abstract: Ada Gem #17 — GNAT warns about functions that might reach their “end”, and sometimes these warnings are false alarms. Pragma No_Return can be used to suppress such false alarms.

Let’s get started…

A function should always return a value or raise an exception. That is, reaching the “end” of a function is an error. In Ada, this error is detected in two ways:

1. Every function must contain at least one return statement. This rule is too liberal, because some control-flow path might bypass the return statement. This rule is too conservative, because it requires a return even when all you want to do is raise an exception:

```
function F (…) return … is
begin
   raise Not_Yet_Implemented;  –  Illegal!
end F;
```

Function F isn’t WRONG — it’s just not finished. In my opinion, this rule should have been abolished, and replaced with a control-flow analysis similar to what GNAT does (see below).

2. If a function reaches its “end”, `Program_Error` is raised. This catches all such errors, but it’s unsatisfying; a compile-time check would be better.

Indeed, GNAT provides an effective compile-time check: it gives a warning if there is any control-flow path that might reach the “end”:

```
function F (…) return … is
begin
   if Some_Condition then
      return Something;
   elsif Some_Other_Condition then
      return Something_Else;
   end if;
   –  CAN get here!
end F;
```

There is a bug, if `Some_Condition` and `Some_Other_Condition` don’t cover all the cases, and GNAT will warn. We can fix it in several ways, such as:

```
function F (…) return … is
begin
   if Some_Condition then
```

```
            return Something;
         elsif Some_Other_Condition then
            return Something_Else;
         else
            raise Some_Exception;
         end if;
      end F;
```

or:

```
      function F (…) return … is
      begin
         if Some_Condition then
            return Something;
         else
            pragma Assert (Some_Other_Condition);
            return Something_Else;
         end if;
      end F;
```

or (and here’s the `No_Return`):

```
   procedure Log_Error (…);
   pragma No_Return (Log_Error);

   function F (…) return … is
   begin
      if Some_Condition then
         return Something;
      elsif Some_Other_Condition then
         return Something_Else;
      else
         Log_Error (…);
      end if;
   end F;
```

GNAT will not cry wolf on the last three versions of F, because it does a simple control-flow analysis to prove that every path reaches either a return statement, or a raise statement, or a non-returning procedure (which presumably contains a raise statement, directly or indirectly).

One way to look at it is that **`pragma`** `No_Return` allows you to wrap a raise statement in a slightly higher level abstraction, without losing the above-mentioned control-flow analysis. `No_Return` is part of the contract; F relies on `Log_Error` not returning, and the compiler ensures that the body of `Log_Error` obeys that contract.

Gem #18: Warnings in GNAT

Author: Bob Duff, AdaCore

Abstract: Ada Gem #18 — This “gem” is not about the Ada language, but about a set of GNAT-specific features related to warnings. We discuss how to make the best use of GNAT’s warnings.

Let’s get started…

The recent “gems” about `pragma No_Return` talked a lot about compile-time checks that generate warnings. Indeed, GNAT is capable of generating lots of useful warnings. Here is some advice about how to get the most out of these warnings.

What’s the difference between a “warning” and an “error”? First, errors are generally violations of the Ada language rules as specified in the Ada Reference Manual; warnings are GNAT specific. Thus, other Ada compilers might not warn about the same things that GNAT does. Second, warnings are typically conservative; that is, some warnings will be false alarms, and the programmer needs to study the code to see if the warning is a real problem.

Some warnings are given by default, whereas some are given only if a switch enables them. Use the -gnatwa switch to turn on (almost) all warnings.

Warnings are useless if you don’t do something about them. If you give your team-member some code that causes warnings, how are they supposed to know if they represent real problems? Pretty soon people will ignore warnings, and they will scatter themselves all about the code. Use the -gnatwae switch to turn on (almost) all warnings, and to treat warnings as errors. This will force you to get a clean (no warnings or errors) compilation.

But some warnings are false alarms. Use `pragma Warnings (Off)` to suppress false alarms. It’s best to be as specific as possible: narrow down to a single line of code, and a single warning message. And use a comment to explain why the warning is a false alarm, if it’s not obvious. The following, compiled with -gnatwae:

```
package body Warnings_Example is

   procedure Mumble (X : Integer) is
   begin
      null;
   end Mumble;

end Warnings_Example;
```

will cause GNAT to complain:

```
warnings_example.adb:5:22: warning: formal parameter "X" is not
referenced
```

But the following will compile cleanly:

```
   package body Warnings_Example is

      pragma Warnings (Off, "formal parameter ""X"" is not
referenced");
      procedure Mumble (X : Integer) is
      pragma Warnings (On, "formal parameter ""X"" is not referenced");
      --  X is ignored here, because blah blah blah...
      begin
         null;
      end Mumble;

   end Warnings_Example;
```

Here we've suppressed the specific warning message on a specific line.

If you get many warnings of a specific type, and it's not feasible to fix them all, then suppress that type of message, so the good warnings won't get buried beneath a pile of bogus ones. The -gnatwaeF switch will silence the warning on the first version of Mumble above: the F means suppress warnings on unreferenced formal parameters, and would be a good idea if you have lots of those.

In summary, I suggest turning on as many warnings as makes sense for your project. Then whenever you see a warning message, look at the code and decide if it's real. If so, fix the code. If it's a false alarm, suppress the warning. Either way, make the warning disappear before checking your code into your configuration management system.

The details of the switches and pragmas can be found in the GNAT Reference Manual and User Guide.

Gem #19: XML streaming of Ada objects

Author: Pascal Obry, EDF R&D

Abstract: Ada Gem #19 — XML streaming of Ada objects

Let’s get started…

Since Ada 95 it has been possible to stream any object. Using `'Input/'Output` or `'Read/'Write` attributes, any object (tagged or not) can be streamed using a binary representation. This means that objects can be written into a file or sent over a socket, for example.

Let’s take a simple object hierarchy to illustrate this feature. We’ll have a Point (x and y coordinate) and a Pixel (a Point with a color).

```
package Object is

   type Point is tagged record
      X, Y : Float;
   end record;

   type Color_Name is (Red, Green, Blue);

   type Pixel is new Point with record
      Color : Color_Name;
   end record;
end Object;
```

When writing a Point or a Pixel the first bytes in the stream are the tag external representation then the object’s attribute values.

```
declare
   File : File_Type;
   P    : Point'Class := ...;
begin
   Create (File, Out_File, "streamed.data");
   Point'Class'Output (Text_Streams.Stream (File), P);
   Close (File);
end;
```

The stream will contain something like (where is the character hexadecimal code):

```
<01> <00> <00> <00> <0C> <00> <00> <00> O B J E C T . P I X E L
<9A> <99> <99> <3f> <66> <66> <06> <41> <00>
```

The tag is an important part as it will be used to be able to create the proper object instance out of the stream.

```
P := constant Point'Class :=
          Point'Class'Input (Text_Streams.Stream (File));
```

All is well! No, there is a little missing feature. There is no way to control how the external tag is streamed. In fact, it is a string and the bounds (lower and upper) are first output into the stream. These bounds are plain numbers written in binary.

In the above example we have the four first bytes for lower bound (equal to 1) and the four following bytes for the upper bound (equal to 12) then the twelve bytes for the external tag full name OBJECT.PIXEL.

In Ada 95 there is no way to stream a textual representation of objects!

But the good news is… Ada 2005 can do this. Ada 2005 goes further by adding support to control finely the external representation of any objects. This means that it is now possible to create a textual representation of such an object using the `'Class'Input` and `'Class'Output` attributes.

Let’s put in place the missing pieces.

First the `'Read` and `'Write` attributes to output or read the XML representation of a Point or a Pixel.

```
   with Ada.Streams;

   package Object is

      type Point is ...

      procedure Read (S : access Root_Stream_Type'Class; O : out
Point);
      for Point'Read use Read;

      procedure Write
        (S : access Root_Stream_Type'Class; O : in Point);
      for Point'Write use Write;

      type Pixel is ...

      procedure Read (S : access Root_Stream_Type'Class; O : out
Pixel);
      for Pixel'Read use Read;

      procedure Write
        (S : access Root_Stream_Type'Class; O : in Pixel);
      for Pixel'Write use Write;
```

The `Read` routines could be implemented using a full featured XML parser like XML/Ada. For conciseness, we will use two very simple XML oriented routines:

```
   procedure Skip_Tag
     (S      : access Ada.Streams.Root_Stream_Type'Class;
      Ending : in     Character := '>');
   --  Skip the next tag on stream S, returns when Ending is found

   function Get_Value
     (S : access Ada.Streams.Root_Stream_Type'Class) return String;
   --  Returns the current value read on stream S
```

Using those routines the `'Read` and `'Write` implementation are straightforward. Here is the implementation for a Point:

```
   procedure Read (S : access Root_Stream_Type'Class; O : out Point) is
   begin
      Skip_Tag (S); O.X := Float'Value (Get_Value (S)); Skip_Tag (S, 
ASCII.LF);
      Skip_Tag (S); O.Y := Float'Value (Get_Value (S)); Skip_Tag (S, 
ASCII.LF);
   end Read;

   procedure Write (S : access Root_Stream_Type'Class; O : in Point) is
   begin
      String'Write (S, "   <x>"  & Float'Image (O.X) & "</x>" & 
ASCII.LF);
      String'Write (S, "   <y>"  & Float'Image (O.Y) & "</y>" & 
ASCII.LF);
   end Write;
```

The last missing piece is the handing of the tag. We want the tag to be simply: <point> and <pixel> (no bound and just the name of the object instead of the full name prefixed by the enclosing package name). To set the proper tag name we use the `External_Tag` attribute:

```
   package Object is

      type Point is ...
      for Point'External_Tag use "point";

      type Pixel is ...
      for Pixel'External_Tag use "pixel";
```

Then we want to plug in our own XML oriented implementation of the `'Class'Input` and `'Class'Output` attributes. This is necessary only for the root type Point:

```
   package Object is

      type Point is ...
      for Point'External_Tag use "point";

      procedure Class_Output
        (S : access Ada.Streams.Root_Stream_Type'Class; O : in 
Point'Class);
      for Point'Class'Output use Class_Output;

      function Class_Input
```

```
        (S : access Ada.Streams.Root_Stream_Type'Class) return
Point'Class;
      for Point'Class'Input use Class_Input;
```

The `Class_Output` routine must output the opening XML tag, output the object itself and then the closing XML tag. Quite simple to do; the following is the commented code:

```
   procedure Class_Output
     (S : access Ada.Streams.Root_Stream_Type'Class; O : in
Point'Class) is
   begin
      --  Write the opening tag
      Character’Write (S, ‘<');
      String'Write (S, Ada.Tags.External_Tag (O'Tag));
      String'Write (S, '>‘ & ASCII.LF);

      –  Write the object, dispatching call to Point/Pixel’Write
      Point’Output (S, O);

      –  Write the closing tag
      String’Write (S, “‘ & ASCII.LF);
   end Class_Output;
```

And now the final part using `Ada.Tags.Generic_Dispatching_Constructor` which will create an object out of a stream given the object’s tag. This must do the exact opposite of the `Class_Output` routine. The opening XML tag is read, then the object using `Generic_Dispatching_Constructor` and finally the closing XML tag.

```
   function Class_Input
     (S : access Ada.Streams.Root_Stream_Type'Class) return Point'Class
   is
      function Dispatching_Input is
         new Ada.Tags.Generic_Dispatching_Constructor
           (T           => Point,
            Parameters  => Ada.Streams.Root_Stream_Type'Class,
            Constructor => Point'Input);
      Input     : String (1 .. 20);
      Input_Len : Natural := 0;
   begin
      --  On the stream we have , we want to get “tag_name”
      –  Read first character, must be ‘<'
      Character'Read (S, Input (1));
      if Input (1) /= '<' then
         raise Ada.Tags.Tag_Error with "starting with " & Input (1);
      end if;

      --  Read the tag name
      Input_Len := 0;
      for I in Input'range loop
         Character'Read (S, Input (I));
         Input_Len := I;
         exit when Input (I) = '>‘;
      end loop;
      —  Check ending tag
      if Input (Input_Len) /= ‘>’
        or else Input_Len <= 1
      then -- Empty tag
         raise Ada.Tags.Tag_Error with "empty tag";
      else
         Input_Len := Input_Len - 1;
      end if;

      declare
         External_Tag : constant String := Input (1 .. Input_Len);
         O            : constant Point'Class := Dispatching_Input
                           (Ada.Tags.Internal_Tag (External_Tag), S);
         --  Dispatches to appropriate Point/Pixel'Input depending on
         --  the tag name.
      begin
         --  Skip closing object tag
         Skip_Tag (S); Skip_Tag (S, ASCII.LF);
         return O;
      end;
   end Class_Input;
```

At this point the code shown at the start will still work without modification. The fact that the object is streamed using an XML representation is transparent to the users of the Object package.

As a final note, for conciseness, the code as-is does not output conformant XML documents as there is no XML header and there are multiple root nodes. This is left as an exercise to the reader.

Gem #20: Using pragma Shared_Passive for data persistence

Author: Pascal Obry, EDF R&D

Abstract: Ada Gem #20 — Using pragma Shared_Passive for data persistence.

Let’s get started…

Data persistence can be achieved in many ways starting as simple as using hand-written code to store and load data into some text files to something as complex as mapping the data into a relational or object database for example.

In some cases we just want to store the content of a set of variables. Ada provides such support using the Annex-E shared passive pragma. Let’s write a simple counter that will increment each time the application is run:

```
package Store is
   pragma Shared_Passive;
   Counter : Natural := 0;
end Store;
```

And yes that’s all! The variable’s current value is read at elaboration time and written during program finalization. Shared passive unit state is saved on disk using one file for each top-level declaration (variables, protected objects). The filename is composed of the unit name and the declaration name separated by a dot. The above counter variable state will be saved into the file named “store.counter” for example.

So the main is as simple as:

```
with Ada.Text_IO;
with Store;

procedure Main is
   use Ada.Text_IO;
begin
   Put_Line (”Counter : ” & Natural’Image (Store.Counter));
   Store.Counter := Store.Counter + 1;
end Main;
```

Each time this program is run it will increment Counter by one. There is nothing to save or load explicitly.

In the context of concurrent programming it may be necessary to add proper synchronization. This can be easily done by using a protected object on the shared passive package.

```
package Store is
   pragma Shared_Passive;
```

```
      protected Shared is
         function Counter return Natural;
         procedure Increment;
      private
         C : Natural := 0;
      end Shared;
   end Store;
```

Note that the set of objects that can be declared is restricted as a shared passive unit can only depend on pure or other shared passive units. So, for example, it is not possible to declare an Unbounded_String nor any Ada.Containers in a shared passive unit.

Yet it is possible to declare complex objects like records or arrays in a shared passive partition and have them automatically saved. Let's take for example the following complex matrix:

```
   package Store is
      pragma Shared_Passive;

      type Complex is record
         X, Y : Float;
      end record;

      type Matrix is array
         (Positive range <>, Positive range <>) of Complex;

      M : Matrix (1 .. 3, 1 .. 3);

   end Store;
```

In spite of the limitations, this cheap persistence support can be quite handy in some circumstances.

Gem #21: How to parse an XML text

Author: Emmanuel Briot, Senior Software Engineer, AdaCore

Abstract: Ada Gem #21 — The World Wide Web Consortium (W3C) develops various specifications around the XML file format. In particular, it specifies various APIs to load, process and write an XML file. Although these APIs are not specified for Ada, XML/Ada tries to conform as closely as possible to them. This gem describes how to use XML/Ada to parse an XML file.

Let's get started…

There are two main APIs to parse an XML file. One (the Document Object Model, DOM) reads the file and generates a tree in memory representing the whole document. Typically, because of the amount of operations mandated by the specifications, this tree is several times larger than the document itself, and thus depending on the amount of memory on your machine, it might limit the size of documents your application can read. On the other hand, it provides a lot of flexibility in the handling of these trees.

The other method (SAX) is based on callbacks, which are called when various constructs are seen while reading the XML file. This requires almost no memory, but makes the processing of the XML file additional work for your application. It is however very well suited when you want to store the XML data in an application-specific data structure. In fact, XML/Ada itself uses SAX to build the DOM tree.

In both cases, XML/Ada needs an object (an "input_source") to read the actual XML data. This data can be found either on the disk, in memory, read from a socket, or any other possible source you can imagine. XML/Ada is carefully constructed so that it doesn't require the whole document in memory, and can just read one character at a time, which makes it adaptable to any possible input. This gem does not cover how to write your own input streams. This is in general quite easy, the only difficulty is to properly convert the bytes you are reading to unicode characters.

Here is a small example on using the DOM API to create a tree in memory. In this example, we are assuming the most frequent case of an XML file on the disk, and therefore we are using a File_Input as the input. The second object we need is the XML parser itself. When we want to create a DOM tree, we need to use a Tree_Reader, or a type derived from it. As we will see later, this is in fact a SAX parser (that is an event-based XML parser) whose callbacks are implemented to create the DOM tree. You can of course override its primitive operations if you want to do additional things (like verbose output, redirect error messages, pre-processing of the XML nodes,…).

```
with Input_Sources.File;  use Input_Sources.File;
with DOM.Readers;         use DOM.Readers;
with DOM.Core;            use DOM.Core;

procedure Read_XML_File (Filename : String) is
```

```
  Input  : File_Input;
  Reader : Tree_Reader;
  Doc    : Document;
begin
  Open (Filename, Input);
  Parse (Reader, Input);
  Close (Input);

  Doc := Get_Tree (Reader);
  ...
  Free (Reader);
end Read_XML_File;
```

The first three lines read the file into memory. The fourth line gets a handle on the tree itself, which you can then manipulate with the various subprograms found in the DOM.Core.* packages (and that are mandated by the W3C specifications). When we are done, we simply free the memory.

There are various settings that can be set on the reader before we actually parse the XML stream, for instance whether it should support XML namespaces, whether we want to validate the input, and so on.

As we mentioned before, there exists a second, lower-level API called SAX which is event-based. It defines one tagged type, a Reader, which has several primitive operations that act as callbacks. You can override the ones you want. In general, the result of calling them is to create an in-memory representation of the XML input (which is what the DOM interface does, really).

The following short example only detects the start of elements in the XML file, and prints their name on standard output. It has little interest in real applications, but is a good framework on which to base your own SAX parsers.

```
with Sax.Attributes;
with Sax.Readers;     use Sax.Readers;
with Unicode.CES;     use Unicode.CES;

package Debug_Parsers is
   type Debug_Reader is new Reader with null record;
   overriding procedure Start_Element
     (Handler       : in out Debug_Reader;
      Namespace_URI : Unicode.CES.Byte_Sequence := "";
      Local_Name    : Unicode.CES.Byte_Sequence := "";
      Qname         : Unicode.CES.Byte_Sequence := "";
      Atts          : Sax.Attributes.Attributes'Class);
end Debug_Parsers;
```

Here is the implementation of the Start_Element callback. We are assuming, in this simple example, that the console on which we are printing the output can accept unicode characters (in fact, all Put_Line does is to print a series of bytes, which are interpreted by the console to do the proper rendering of unicode glyphs).

```
with Ada.Text_IO;   use Ada.Text_IO;

package body Debug_Parsers is
   procedure Start_Element
     (Handler       : in out Debug_Reader;
      Namespace_URI : Unicode.CES.Byte_Sequence := "";
      Local_Name    : Unicode.CES.Byte_Sequence := "";
      Qname         : Unicode.CES.Byte_Sequence := "";
      Atts          : Sax.Attributes.Attributes'Class)
   is
   begin
      Put_Line ("Found start of " & Qname);
   end Start_Element;
end Debug_Parsers;
```

And finally here is a short example of a program using that parser. Notice how it closely mimics what we did for DOM (which is not so surprising, since, once again, the DOM parser itself is really a special implementation of a SAX parser).

```
with Input_Sources.File;  use Input_Sources.File;
with Debug_Parsers;       use Debug_Parsers;

procedure Test_Sax is
  Input  : File_Input;
  Reader : Debug_Reader;
begin
  Open (Filename, Input);
  Parse (Reader, Input);
  Close (Input);
end Test_Sax;
```