Gem #22: Ada Speaks Many Languages

Author: Robert Dewar, Vasiliy Fofanov, Franco Gasperoni, Yang Zhang

Abstract: Ada Gem #22 — Ada speaks many languages. How many? All those languages that you can write with a computer keyboard. Read on to learn about this Ada gem.

Let's get started…

The only characters allowed in an Ada 83 program (for strings, comments, and identifiers) were 7-bit ASCII symbols. That was annoying. In Ada 83 we could not write:

```
S : String : = "à la carte";
```

and even less write:

```
À_La_Carte : Boolean := False;
```

That was - how shall we put it - a "set menu" - view of things :)

An amendment changed this situation to 8-bit characters during the lifetime of Ada 83. Ada 95 made this 8-bit change clearer and more official, designating ISO Latin-1 as the character set. So the above are both legal in Ada 95. Ada 95 also introduced 16-bit ISO 10646 support in the form of Wide_Character. One can write in Ada 95:

```
My_Favorite_Pie : String := "π";   --  :)
```

but an implementation did not have to allow 16-bit characters in identifiers and comments, and Ada 95 did not mandate the acceptance of the following:

```
    π : constant :=
3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37511;
    —  Can't eat this one :)
```

although the GNAT technology for Ada 95 allows full 16-bit characters in identifiers and comments.

Ada 2005 is the ultimate in terms of openness: the full 32-bit ISO-10646 character set is supported and use of π (for instance) in identifiers, comments, and strings is allowed in an Ada 2005 program. As a matter of fact the package Ada.Numerics in Ada 2005 now contains:

```
  Pi : constant :=
3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37511;
  π : constant := Pi;
```

To demonstrate the use of the full 32-bit ISO-10646 character set in Ada 2005 programs we have written a couple of programs in English, Russian, and Chinese. These programs take an ISO date ranging from 1983 to 2019 and print the date in the local format. Each program wishes you a happy new year if the date entered matches the local new year date. All file names are of the form:

```
happy_new_year_{locale}_{lang}.adb
```

where `{locale}` is a 2 letter sequence indicating the country for which the program has been written (e.g. “us” for the US, “cn” for China, “ru” for Russia) and `{lang}` the language in which the program has been written (e.g. “en” for English, “cn” for Chinese, “ru” for Russian). For instance “happy_new_year_cn_en.adb” is the program for China written in English, while “happy_new_year_cn_cn.adb” is the program for China written in Chinese. Having a program in English for China allows non Chinese speakers to understand what the program does and perhaps learn some Chinese :)

To compile the Ada programs provided with this gem we suggest that you use options `-gnatW8 -gnat05`.

Enjoy… Happy Holidays and Happy New Year :)

Related Source Code

The files referenced in this gem are available at the website. Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #23: Null Considered Harmful

Author: Bob Duff, AdaCore

Abstract: Ada Gem #23 — The "not null" syntax allows an Ada 2005 program to prevent access values from being null in cases where the null value is undesirable. This new syntax helps provide useful documentation**.**

Let's get started…

Ada, like many languages, defines a special 'null' value for access types. All values of an access type designate some object of the designated type, except for null, which does not designate any object. The null value can be used as a special flag. For example, a singly-linked list can be null-terminated. A Lookup function can return null to mean "not found", presuming the result is of an access type:

```
type Ref_Element is access all Element;
Not_Found : constant Ref_Element := null;
function Lookup (T : Table) return Ref_Element;
-  Returns Not_Found if not found.
```

An alternative design for Lookup would be to raise an exception:

```
Not_Found : exception;
function Lookup (T : Table) return Ref_Element;
—  Raises Not_Found if not found.
-  Never returns null.                    <-  Ada 95 comment.
```

Neither design is better in all situations; it depends in part on whether we consider the "not found" situation to be exceptional.

Clearly, the client calling Lookup needs to know whether it can return null, and if so, what that means. In general, it's a good idea to document whether things can be null or not, especially for formal parameters and function results. In Ada 95, we do that with comments. In Ada 2005, we can use the "not null" syntax:

```
function Lookup (T : Table) return not null Ref_Element;  -  Ada
2005
```

In general, it's better to use the language proper for documentation, when possible, rather than comments, because compile-time and/or run-time checks can help ensure that the "documentation" is actually true. With comments, there's a greater danger that the comment will become false during maintenance, and false documentation is obviously a menace.

In many, perhaps most, cases, null is just a tripping hazard. It's a good idea to put in "not null" when possible. In fact, a good argument can be made that "not null" should be the default, with extra syntax required when null *is* wanted. This is the way SML works, for

example — you don't get any special null-like value unless you ask for it. Of course, Ada 2005 needs to be compatible with Ada 95, so "not null" cannot be the default for Ada.

One word of caution: access objects are default-initialized to null, so if you have a "not null" object (or component) you had better initialize it explicitly, or you will get Constraint_Error. "Not null" is more often useful on parameters and function results, for this reason.

Gem #24: Null Considered Harmful (Part 2 — Efficiency)

Author: Bob Duff, AdaCore

Abstract: Ada Gem #24 — The "not null" syntax can make programs more efficient by removing the need for implicit run-time checks.

Let's get started…

In last week's gem, we talked about the documentation advantages of using "not null". Here's another example, first with null:

```
procedure Iterate
  (T : Table;
   Action : access procedure (X : not null Ref_Element)
                                    := null);
-  If Action is null, do nothing.
```

…and without null:

```
procedure Do_Nothing (X : not null Ref_Element) is null;
procedure Iterate
  (T : Table;
   Action : not null access procedure (X : not null Ref_Element)
                                    := Do_Nothing'Access);
```

I much prefer the style of the second Iterate.

The "not null access procedure" is quite a mouthful, but it's worthwhile, and anyway, the compatibility requirement for Ada 2005 requires that the "not null" be explicit, rather than the other way around.

Another advantage of "not null" over comments is for efficiency. For example:

```
procedure P (X : not null Ref_Element) is
begin
   X.all.Component := X.all.Component + 1;
end P;

procedure Q (X : not null Ref_Element) is
begin
   while … loop
      P (X);
   end loop;
end Q;

procedure R is
begin
   Q (An_Element'Access);
end R;
```

Without “not null”, the generated code for P will do a check that X /= null, which may be costly on some systems. P is called in a loop, so this check will likely occur many times. With “not null”, the check is pushed to the call site. Pushing checks to the call site is usually beneficial because (1) the check might be hoisted out of a loop by the optimizer, or (2) the check might be eliminated altogether, as in the example above, where the compiler knows that An_Element’Access cannot be null.

This is analogous to the situation in Ada 95 with other run-time checks, such as array bounds checks:

```
type My_Index is range 1..10;
type My_Array is array (My_Index) of Integer;

procedure Process_Array (X : in out My_Array; Index : My_Index);
```

If “X (Index)” occurs inside Process_Array, there is no need to check that Index is in range, because the check is pushed to the caller.

Gem #25: How to Search Text

Author: Emmanuel Briot, Senior Software Engineer, AdaCore

Abstract: Ada Gem #25 — The Ada standard defines several subprograms related to searching text in a string. In addition, GNAT provides additional packages to search. This gem will cover the various possibilities.

Let's get started…

The Ada standard defines several subprograms related to searching text in a string. In addition, GNAT provides additional packages to search. This gem will cover the various possibilities.

We are assuming that we are searching for text within a string that is in memory. To search text within a file, the simplest is to load the file into memory, and then search the resulting string.

We are also assuming we are dealing with a `String` type, not with an `Unbounded_String` or `Bounded_String`. In both cases, one can easily convert to a string. Predefined Ada subprograms exist for all three of these types, but the GNAT specific packages deal only with Strings. Of course, converting to a string is not necessarily efficient, and one should point to the GNAT-specific packages `Ada.Strings.Unbounded.Aux` which allow you to peek inside the unbounded string, but should obviously be used with care.

The text we are searching for (the pattern) can have various forms, however. It can be a fixed text which must be found exactly as is, or it can be a regular expression which matches a whole set of strings.

When searching for fixed text, the obvious candidates are the predefined Ada runtime subprograms from the `Ada.Strings.Fixed` package. The relevant subprograms all start with the prefix Index. These are reasonably optimized, even though they are not written specifically in assembly for maximum speed. Similar subprograms exist for bounded and unbounded strings.

Some advanced algorithms exist for searching fixed text within very long texts. These algorithms spend a bit of time studying the pattern initially, creating an internal representation (for instance a finite-state automaton), so that later searches are more efficient. One example of this is the Boyer-Moore algorithm. The initial computation can be more costly than the time it takes to do the search when the string is small, so using Index in such cases is the most efficient solution. However, as the size of the string grows, using a more advanced algorithm becomes interesting. The GNAT runtime does not currently provide such an implementation, but the GPS IDE has such a package which will likely be integrated into the GNAT runtime in the future. Let us know of your interest in such a package!

Regular expressions are a well known feature in several programming languages like Perl, and provide advanced search capabilities. They provide a syntax for describing a pattern that should be matched. For instance, “a.*b” would find all substrings starting with “a” and ending with “b”, with zero or more characters in between. Likewise, “a[bc]d” would find all three-characters substrings starting with “a”, ending with “d”, and whose middle character is either “b” or “c”. This gem is not a tutorial on how to write regular expressions, since for instance the Perl manual or various tutorials on the internet already cover the topic at length.

GNAT provides two packages for searching for regular expressions.

The one which has been around for longest is GNAT.Regexp. It is not a fully-general regular expression package, since it does not support, for instance, retrieving the substrings matched by parenthesis groups. It also tries to match the whole text; that is, “a[bc]d” will only match if the text itself is three characters long. It supports two versions of the regular expressions, the usual ones as described above and what are generally called “globbing” patterns, as used by the various command line shells on Unix. Therefore, GNAT.Regexp is most often used for searches related to file names.

Here is a short example on how to find all Ada source files in the current directory. As you will notice, the regular expression first needs to be compiled into a finite state automaton, and then provides very efficient pattern matching through the Match function.

```
with Ada.Directories; use Ada.Directories;
with GNAT.Regexp;     use GNAT.Regexp;

procedure Search_Files (Pattern : String) is
  Search : Search_Type;
  Ent    : Directory_Entry_Type;
  Re     : constant Regexp := Compile (Pattern, Glob => True);

begin
  Start_Search (Search, Directory => ".", Pattern => "");
  while More_Entries (Search) loop
     Get_Next_Entry (Search, Ent);
     if Match (Simply_Name (Ent), Re) then
         … —  Matched, do whatever
     end if;
  end loop;
  End_Search (Search);
end Search_Files;

Search_Files ("*.adb");
```

A second package that GNAT provides is GNAT.Regpat. It provides full handling of the usual regular expression constructs, but not some of the recent advances that languages like Perl have seen (look-ahead, unicode,…). A regular expression is first compiled into a byte-code which will provide faster matching. Note that this package is not as efficient as GNAT.Regexp, since some patterns will read characters, discover they do not match,

then backtrack to an earlier position and try some other possibility. In practice, the efficiency is generally good enough.

One feature that this package provides over GNAT.Regexp is the capability, after a match, to know what parenthesis groups matched. For instance, in the regular expression "`a (big|small) application`", it is possible to know which of the two alternatives was matched, as in the following example:

```
with GNAT.Regpat;   use GNAT.Regpat;
with Ada.Text_IO;   use Ada.Text_IO;

procedure Search_Regexp (Pattern : String; Search_In : String) is
  Re      : constant Pattern_Matcher := Compile (Pattern);
  Matches : Match_Array (0 .. 1);

begin
  Match (Re, Search_In, Matches);
  if Matches (0) = No_Match then
     Put_Line ("The pattern did not match");
  else
     Put_Line ("The application really is "
               & Search_In (Matches (1).First .. Matches (1).Last));
  end if;
end Search_Regexp;

Search_Regexp ("a (big|small) application",
               "Example of a small application for searching");
```

Although regular expressions are quite powerful, and generally more than enough for most applications, they have limitations that only a full context free grammar can overcome. One classical example is that they cannot ensure that a string has the same number of opening and closing parenthesis, as in "(a(b))", when the number of parentheses is not known in advance.

GNAT provides the package GNAT.Spitbol.Patterns, which contains various subprograms that can be used to implement matching with context free grammars. SPITBOL itself is a full programming language, which GNAT can emulate through the subprograms in GNAT.Spitbol, but in this gem we only concentrate on the pattern matching capabilities.

Here is a small example on using GNAT.Spitbol.Patterns. The goal here is to check whether the given string starts with a balanced expression with respect to "[" and "{", that is if the first character is one of those, then there must be a similar closing character, and the substring in between is also properly balanced.

The language for such a pattern in BNF form would be:

```
  ELEMENT ::= <any character other than [] or {}>
              | '[' BALANCED_STRING ']'
              | '{' BALANCED_STRING '}'
```

```
  BALANCED_STRING ::= ELEMENT {ELEMENT}
```

which gets translated to the following spitbol program (see the documentation in GNAT.Spitbol.Patterns for more information on the format of the patterns). Note that this program does not force the end of the string to be properly balanced, just the beginning of it.

```
with GNAT.Spitbol.Patterns;  use GNAT.Spitbol.Patterns;

procedure Search_Spitbol is
  Element, Balanced_String : aliased Pattern;
  At_Start : Pattern;

begin
  Element := NotAny ("[]{}")
        or ('[' & (+Balanced_String) & ']')
        or ('{' & (+Balanced_String) & '}');
  Balanced_String := Element & Arbno (Element);
  At_Start := Pos (0) & Balanced_String;
  Match ("[ab{cd}]",   At_Start); --  True
  Match ("[ab{cd}",    At_Start); --  False
  Match ("ab{cd}",     At_Start); --  True
end Search_Spitbol;
```

Gem #26: The Mod Attribute

Author: Bob Duff, AdaCore

Abstract: Ada Gem #26 — T'Mod can be used to convert signed integers to modular integers using modular (wraparound) arithmetic.

Let's get started…

Ada has two kinds of integer type: signed and modular:

```
type Signed_Integer is range 1..1_000_000;
type Modular is mod 2**32;
```

Operations on signed integers can overflow: if the result is outside the base range, `Constraint_Error` will be raised. The base range of `Signed_Integer` is the range of `Signed_Integer'Base`, which is chosen by the compiler, but is likely to be something like `-2**31..2**31-1`.

Operations on modular integers use modular (wraparound) arithmetic.

For example:

```
X : Modular := 1;
X := - X;
```

Negating X gives -1, which wraps around to `2**32-1`, i.e. all-one-bits.

But what about a type conversion from signed to modular? Is that a signed operation (so it should overflow) or is it a modular operation (so it should wrap around)? The answer in Ada is the former — that is, if you try to convert, say, `Integer'(-1)` to Modular, you will get `Constraint_Error`:

```
I : Integer := -1;
X := Modular (I);  -  raises Constraint_Error
```

In Ada 95, the only way to do that conversion is to use `Unchecked_Conversion`, which is somewhat uncomfortable. Furthermore, if you're trying to convert to a generic formal modular type, how do you know what size of signed integer type to use? Note that `Unchecked_Conversion` might malfunction if the source and target types are of different sizes.

A small feature added to Ada 2005 solves the problem: the `Mod` attribute:

```
generic
   type Formal_Modular is mod <>;
package Mod_Attribute is
```

```
   function F return Formal_Modular;
end Mod_Attribute;

package body Mod_Attribute is

   A_Signed_Integer : Integer := -1;

   function F return Formal_Modular is
   begin
      return Formal_Modular'Mod (A_Signed_Integer);
   end F;

end Mod_Attribute;
```

The `Mod` attribute will correctly convert from any integer type to a given modular type, using wraparound semantics. Thus, F will return the all-ones bit pattern, for whatever modular type is passed to `Formal_Modular`.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #27: Changing Data Representation (Part 1)

Author: Robert Dewar, AdaCore

Abstract: **Ada Gem #27 — Part 1, Automatic Representation Changes**

Let's get started...

A powerful feature of Ada is the ability to specify the exact data layout. This is particularly important when you have an external device or program that requires a very specific format. Some examples are:

```
type Com_Packet is record
   Key : Boolean;
   Id  : Character;
   Val : Integer range 100 .. 227;
end record;

for Com_Packet use record
   Key at 0 range 0 .. 0;
   Id  at 0 range 1 .. 8;
   Val at 0 range 9 .. 15;
end record;
```

which lays out the fields of a record, and in the case of Val, forces a biased representation in which all zero bits represents 100. Another example is:

```
type Val is (A,B,C,D,E,F,G,H);
type Arr is array (1 .. 16) of Val;
for Arr'Component_Size use 3;
```

which forces the components to take only 3 bits, crossing byte boundaries as needed. A final example is:

```
type Status is (Off, On, Unknown);
for Status use (Off => 2#001#, On => 2#010#, Unknown => 2#100#);
```

which allows specified values for an enumeration type, instead of the efficient default values of 0,1,2.

In all these cases, we might use these representation clauses to match external specifications, which can be very useful. The disadvantage of such layouts is that they are inefficient, and accessing individual components, or in the case of the enumeration type, looping through the values, can increase space and time requirements for the program code.

One approach that is often effective is to read or write the data in question in this specified form, but internally in the program represent the data in the normal default

layout, allowing efficient access, and do all internal computations with this more efficient form.

To follow this approach, you will need to convert between the efficient format and the specified format. Ada provides a very convenient method for doing this, as described in RM 13.6 “Change of Representation”.

The idea is to use type derivation, where one type has the specified format and the other has the normal default format. For instance for the array case above, we would write:

```
type Val is (A,B,C,D,E,F,G,H);
type Arr is array (1 .. 16) of Val;

type External_Arr is new Arr;
for External_Arr'Component_Size use 3;
```

Now we read and write the data using the External_Arr type. When we want to convert to the efficient form, Arr, we simply use a type conversion.

```
Input_Data  : External_Arr;
Work_Data   : Arr;
Output_Data : External_Arr;

(read data into Input_Data)

--  Now convert to internal form
Work_Data := Arr (Input_Data);

(computations using efficient Work_Data form)

—  Convert back to external form
Output_Data := External_Arr (Work_Data);
```

Using this approach, the quite complex task of copying all the data of the array from one form to another, with all the necessary masking and shift operations, is completely automatic.

Similar code can be used in the record and enumeration type cases. It is even possible to specify two different representations for the two types, and convert from one form to the other, as in:

```
type Status_In is (Off, On, Unknown);
type Status_Out is new Status_In;

for Status_In use (Off => 2#001#, On => 2#010#, Unknown => 2#100#);
for Status_Out use (Off => 103, On => 1045, Unknown => 7700);
```

There are two restrictions that must be kept in mind when using this feature. First, you have to use a derived type. You can't put representation clauses on subtypes, which means that the conversion must always be explicit. Second, there is a rule RM 13.1(10) that restricts the placement of interesting representation clauses:

10 For an untagged derived type, no type-related representation items are allowed if the parent type is a by-reference type, or has any user-defined primitive subprograms.

All the representation clauses that are interesting from the point of view of change of representation are "type related", so for example, the following sequence would be illegal:

```
type Val is (A,B,C,D,E,F,G,H);
type Arr is array (1 .. 16) of Val;

procedure Rearrange (Arg : in out Arr);

type External_Arr is new Arr;
for External_Arr'Component_Size use 3;
```

Why these restrictions? Well the answer is a little complex, and has to do with efficiency considerations, which we will address in next week's GEM.

Gem #28: Changing Data Representation (Part 2)

Author: Robert Dewar, AdaCore

Abstract: Ada Gem #28 — Part 2, Efficiency Considerations

Last week, we discussed the use of derived types and representation clauses to achieve automatic change of representation. More accurately, this feature is not completely automatic, since it requires you to write an explicit conversion. In fact there is a principle behind the design here which says that a change of representation should never occur implicitly behind the back of the programmer without such an explicit request by means of a type conversion.

The reason for that is that the change of representation operation can be very expensive, since in general it can require component by component copying, changing the representation on each comoponent.

Let's have a look at the -gnatG expanded code to see what is hidden under thecovers here. For example, the conversion Arr (Input_Data) from last week's example generates the following expanded code:

```
   B26b : declare
      [subtype p__TarrD1 is integer range 1 .. 16]
      R25b : p__TarrD1 := 1;
   begin
      for L24b in 1 .. 16 loop
         [subtype p__arr___XP3 is
           system__unsigned_types__long_long_unsigned range 0 ..
           16#FFFF_FFFF_FFFF#]
         work_data := p__arr___XP3!((work_data and not shift_left!(
           16#7#, 3 * (integer(L24b - 1)))) or
shift_left!(p__arr___XP3!
           (input_data (R25b)), 3 * (integer(L24b - 1))));
         R25b := p__TarrD1'succ(R25b);
      end loop;
   end B26b;
```

That's pretty horrible! In fact one of the Ada experts here thought that it was too gruesome and suggested simplifying it for this gem, but we have left it in its original form, so that you can see why it is nice to let the compiler generate all this stuff so you don't have to worry about it yourself.

Given that the conversion can be pretty inefficient, you don't want to convert backwards and forwards more than you have to, and the whole approach is only worth while if will be doing extensive computations involving the value.

The expense of the conversion explains two aspects of this feature that are not obvious. First, why do we require derived types instead of just allowing subtypes to have different representations, avoiding the need for an explicit conversion?

The answer is precisely that the conversions are expensive, and you don't want them happening behind your back. So if you write the explicit conversion, you get all the gobbledygook listed above, but you can be sure that this never happens unless you explicitly ask for it.

This also explains the restriction we mentioned in last week's gem from RM 13.1(10):

10 For an untagged derived type, no type-related representation items are allowed if the parent type is a by-reference type, or has any user-defined primitive subprograms.

It turns out this restriction is all about avoiding implicit changes of representation. Let's have a look at how type derivation works when there are primitive subprograms defined at the point of derivation. Connsider this example:

```
type My_Int_1 is range 1 .. 10;

function Odd (Arg : My_Int_1) return Boolean;

type My_Int_2 is new My_Int_1;
```

Now when we do the type derivation, we inherit the function Odd for My_Int_2. But where does this function come from? We haven't written it explicitly, so the compiler somehow materializes this new implicit function. How does it do that?

We might think that a complete new function is created including a body in which My_Int_2 replaces My_Int_1, but that would be impractical and expensive. The actual mechanism avoids the need to do this by use of implicit type conversions. Suppose after the above declarations, we write:

```
Var : My_Int_2;
...
if Odd (Var) then
   …
```

The compiler translates this as:

```
Var : My_Int_2;
...
if Odd (My_Int_1 (Var)) then
   …
```

This implicit conversion is a nice trick, it means that we can get the effect of inheriting a new operation without actually having to create it. Furthermore, in a case like this, the type conversion generates no code, since My_Int_1 and My_Int_2 have the same representation.

But the whole point is that they might not have the same representation if one of them had a rep clause that made the representations different, and in this case the implicit conversion inserted by the compiler could be expensive, perhaps generating the junk we quoted above for the Arr case. Since we never want that to happen implicitly, there is a rule to prevent it.

The business of forbidding by-reference types (which includes all tagged types) is also driven by this consideration. If the representations are the same, it is fine to pass by reference, even in the presence of the conversion, but if there was a change of representation, it would force a copy, which would violate the by-reference requirement.

So to summarize these two gems, on the one hand Ada gives you a very convenient way to trigger these complex conversions between different representations. On the other hand, Ada guarantees that you never get these potentially expensive conversions happening unless you explicitly ask for them.

Gem #29: Introduction to the Ada Web Server (AWS)

Author: Pascal Obry, EDF R&D

Abstract: Ada Gem #29 — Introduction to Ada Web Server (AWS)

Let's get started…

This Gem presents a basic introduction the Ada Web Server (AWS). The core components of AWS comprise a Web server which supports the HTTP protocol. This Web server can be embedded into any application. Common usages are:

- To develop a full Web application.
- Adding a GUI to a mostly batch-oriented application like a long-running simulation that we want to get control of from time to time.
- To develop a distributed application exchanging messages using the HTTP or SOAP protocols.

The AWS HTTP module handles the decoding of the HTTP request and encoding of the user's response. It uses a callback mechanism to interact with the user's application. Let's take the famous Hello World example and see how it could be translated into the AWS framework.

First the user's hello_world callback, which is a function with a single parameter of type Status.Data and returning a Response.Data object:

```
with AWS.MIME;
with AWS.Response;
with AWS.Status;

package body CB is
   use AWS;
   function Hello_World (Request : in Status.Data)
      return Response.Data
   is
   begin
      return Response.Build (MIME.Text_HTML, "<p>Hello World!</p>");
   end Hello_World;
end CB;
```

The AWS.Response unit contains many constructors. Response.Build is one. Another is Response.File, for returning a file. Various high-level constructors are provided, such as for streaming data.

Now here's the main subprogram with the server:

```
with AWS.Server;
with CB;
procedure Hello_World is
   use AWS;
   HTTP : Server.HTTP;
begin
   Server.Start (HTTP,
                 "Hello_World",
                 Callback => CB.Hello_World'Access);
   delay 60.0;
   Server.Shutdown (HTTP);
end Hello_World;
```

That's it. The start routine registers the CB.Hello_World procedure as the user's callback and starts the server using the default port, which is 8080. After running the hello_world executable, by entering the URL http://localhost:8080/ into your web browser you'll receive the message "Hello World", and this will occur for any URI. So entering http://localhost:8080/whatever will also return "Hello World". After 60 seconds the server will shut down and the program will exit.

All HTTP parameters are available from the Request parameter of the Hello_World callback. So let's add a local constant to get the actual URI received:

```
URI : constant String := Status.URI (Request);
```

Then we change the Response.Build call to:

```
return Response.Build
   (MIME.Text_HTML, "<p>Hello World! URI=" & URI & "</p>");
```

After restarting the server, entering http://localhost:8080/home into the web browser will display "Hello World! URI=/home".

An if/elsif construction can be used to test for each URI that must be handled by the server:

```
if URI = "…" then
   …
elsif URI = "…" then
   …
else
   return Response.Build
      (MIME.Text_HTML,
       "<p>Not found : URI=" & URI & "</p>",
       Status_Code => Messages.S404);
end if;
```

Using callbacks is simple, but when the application becomes larger it's easier to use dispatchers. So let's change the Hello World program to use dispatchers:

```
with AWS.Response;
with AWS.Status;
with AWS.Dispatchers;
package CB is
   use AWS;

   type Hello_World is new Dispatchers.Handler with null record;

   overriding function Dispatch
     (Handler : in Hello_World;
      Request : in Status.Data) return Response.Data;
private
   overriding
   function Clone (Element : in Hello_World) return Hello_World;
end CB;
```

The body of Dispatch is identical to the Hello_World callback above. The Dispatchers.Handler type has a clonable interface, and Clone in this case is trivial:

```
overriding
function Clone (Element : in Hello_World) return Hello_World is
begin
   return Element;
end Clone;
```

In this case the main subprogram is slightly larger:

```
with AWS.Config;
with AWS.Server;
with AWS.Services.Dispatchers.URI;
with CB;

procedure Hello_World is
   use AWS;
   HTTP : Server.HTTP;
   Conf : Config.Object;
   HW   : CB.Hello_World;
   Root : Services.Dispatchers.URI.Handler;
begin
   Services.Dispatchers.URI.Register (Root, "/hello", HW);
   Server.Start (HTTP, Root, Conf);
   delay 60.0;
   Server.Shutdown (HTTP);
end Hello_World;
```

The main difference is that we have registered the HW dispatcher to be used only for URI "/hello" (http://localhost:8080/hello). For any other URL, a 404 error message will be sent by the dispatcher module. It's also possible to use a regular expression or a prefix if

needed. The configuration object can be set to change any server settings, such as the port, the number of simultaneous connections, timeouts, etc.

Many kinds of dispatchers exist, not only for supporting URIs (as in this example), but also for handling virtual hosts, request methods (POST/GET) and services for linking dispatchers, as well as dispatching based on timers (see child units of AWS.Services.Dispatchers). For an example of a simple API for converting a callback to a dispatcher see AWS.Dispatchers.Callback.

Various other services are offered directly by AWS, including HTTP/SOAP, WSDL, Ajax, SMTP, and a templates engine. But covering all of those is beyond the scope of this introduction.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #30: Safe and Secure Software : Introduction

Author: John Barnes

Abstract: This week's gem is the introduction to John Barnes' new booklet:

Safe and Secure Software: An Introduction to Ada 2005.

Over the coming months, we will be publishing all thirteen chapters of the booklet. In the attachment at the bottom of this page you can access the contents and bibliography for the entire booklet. We hope you will enjoy the read!

Let's get started…

The aim of this booklet is to show how Ada 2005 addresses the needs of designers and implementers of safe and secure software. The discussion will also show that those aspects of Ada that make it ideal for safety-critical and security-critical application areas will also simplify the development of robust and reliable software in many other areas.

The world is becoming more and more concerned about both safety and security. Moreover, software now pervades all aspects of the workings of society. Accordingly, it is important that software which is concerned with systems for which safety or security are a major concern should be safe and secure.

There has been a long tradition of concern for safety going back to the development of railroad signaling and more recently with aviation. Vital software systems such as those that control aircraft navigation and landing have to meet well established certification and validation criteria.

More recently there has been growing concern with security in systems such as banking and communications generally. This has been heightened with concern for the activities of terrorists.

Safety and security are intertwined through communication. An interesting characterization of the difference is
- safety – the software must not harm the world,
- security – the world must not harm the software.

So a safety-critical system is one in which the program must be correct, otherwise it might wrongly change some external device such as an aircraft flap or a railroad signal, with serious real-world consequences.

And a security-critical system is one in which it must not be possible for some incorrect or malicious input from the outside to violate the integrity of the system, for example by corrupting a password checking mechanism and stealing social security information.

The key to guarding against both problems is that the software must be correct in the aspects affecting the system's integrity. And by correct we mean that it meets its specification. Of course if the specification is incomplete or itself incorrect then the system will be vulnerable. Capturing requirements correctly is a hard problem and is the focus of much attention from the lean software development community.

One of the trends of the second half of the twentieth century was a universal concern with freedom. But there are two aspects of freedom. The ability of the individual to do whatever they want conflicts with the right to be protected from the actions of others. Maybe A would like the freedom to smoke in a pub whereas B wants freedom from smoke in a pub. Concern with health in this example is changing the balance between these freedoms. Maybe the twenty-first century will see further shifts from "freedom to" to "freedom from".

In terms of software, the languages Ada and C have very different attitudes to freedom. Ada introduces restrictions and checks, with the goal of providing freedom from errors. On the other hand C gives the programmer more freedom, making it easier to make errors.

One of the historical guidelines in C was "trust the programmer". This would be fine were it not for the fact that programmers, like all humans, are frail and fallible beings. Experience shows that whatever techniques are used it is hard to write "correct" software. It is good advice therefore to use tools that can help by finding bugs and preventing bugs. Ada was specifically designed to help in this respect. There have been three versions of Ada – Ada 83, Ada 95 and now Ada 2005.

The purpose of this booklet is to illustrate the ways in which Ada 2005 can help in the construction of reliable software, by illustrating some aspects of its features. It is hoped that it will be of interest to programmers and managers at all levels.

It must be stressed that the discussion is not complete. Each chapter selects a particular topic under the banner of Safe X where Safe is just a brief token to designate both safety and security. For the most critical software, use of the related SPARK language appears to be very beneficial, and this is outlined in Chapter 11.

A topic with which Ada has much synergy is lean software development – there is not enough space in this booklet to expand on this concept but the reader is encouraged to explore its good ideas elsewhere.

As the twenty-first century progresses we will see software becoming even more pervasive. It would be nice to think that software in automobiles for example was developed with the same care as that in airplanes. But that is not so. My wife recently had an experience where her car displayed two warning icons. One said "stop at once", the other said "drive immediately to your dealer". Another anecdotal motor story is that of a driver attempting to select channel 5 on the radio, only to see the car change into 5th gear! Luckily he did not try Replay.

For a fuller description of Ada 2005, SPARK, and lean software development and papers on related topics please consult the bibliography.

Read on…

Note: All chapters will also be available on the Ada 2005 home page.