Gem #31: preconditions/postconditions

Author: Robert Dewar

Abstract: Ada Gem #31 — The notion of preconditions and postconditions is an old one. A precondition is a condition that must be true before a section of code is executed, and a postcondition is a condition that must be true after the section of code is executed.

Let's get started…

The notion of preconditions and postconditions is an old one. A precondition is a condition that must be true before a section of code is executed, and a postcondition is a condition that must be true after the section of code is executed.

In the context we are talking about here, the section of code will always be a subprogram. Preconditions are conditions that must be guaranteed by the caller before the call, and postconditions are results guaranteed by the subprogram code itself.

It is possible, using pragma Assert (as defined in Ada 2005, and as implemented in all versions of GNAT), to approximate run-time checks corresponding to preconditions and postconditions by placing assertion pragmas in the body of the subprogram, but there are several problems with that approach:

1. The assertions are not visible in the spec, and preconditions and postconditions are logically a part of (in fact, an important part of) the spec.

2. Postconditions have to be repeated at every exit point.

3. Postconditions often refer to the original value of a parameter on entry or the result of a function, and there is no easy way to do that in an assertion.

The latest versions of GNAT implement two pragmas, Precondition and Postcondition, that deal with all three problems in a convenient way. The easiest way to describe these is to use an example:

```
package Arith is
   function Sqrt (Arg : Integer) return Integer;
   pragma Precondition (Arg >= 0);
   pragma Postcondition
    (Sqrt’Result >= 0
       and then
    (Sqrt’Result ** 2) <= Arg
       and then
    (Sqrt’Result + 1) ** 2 > Arg);
end Arith;
```

```
with Arith; use Arith;
with Text_IO; use Text_IO;
procedure Main is
begin
   Put_Line (Sqrt (9)'Img);
   Put_Line (Sqrt (10)'Img);
   Put_Line (Sqrt (-3)'Img);
end;
```

Now if we compile with -gnata (which enables preconditions and postconditions), and we have a correct body for Sqrt, then when we run Main we will get:

```
 3
 3

raised SYSTEM.ASSERTIONS.ASSERT_FAILURE :
   Precondition failed at arith.ads:3
```

Now if there was something wrong with the body of Sqrt that gave the wrong answer, we might get:

```
raised SYSTEM.ASSERTIONS.ASSERT_FAILURE :
  postcondition failed at arith.ads:4
```

Indicating that we have a bug in the body of Sqrt that we must investigate.

There is one more thing to mention, which is the promised ability to refer to the old value of parameters. A new attribute 'Old allows this as shown in this example:

```
procedure Write_Data (Total_Writes : in out Natural);
--  Write out the data incrementing Total_Writes to
--  show number of write operations performed.
pragma Postcondition (Total_Writes > Total_Writes'Old);
```

The introduction of preconditions and postconditions into GNAT provides a powerful tool for design and documentation (Eiffel has referred to this approach as "design by contract").

The preconditions and postconditions serve three functions
1. They provide valuable formal documentation in the spec
2. They provide input to proof tools
3. They help find bugs in the course of normal debugging as shown by the example above.

Support for preconditions and postconditions will be available in GNAT Pro 6.2.1 and forthcoming versions of GNAT GPL. If you are a GNAT Pro user and you want to try this feature out today, request a wavefront by using GNAT Tracker.

Gem #32: Safe and Secure Software : Chapter 1, Safe Syntax

Author: John Barnes

Abstract: This week's gem is the first chapter of 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 Gem #30 you can access the contents and bibliography for the entire booklet. We hope you will enjoy the read!

Let's get started…

Syntax is often considered to be a rather boring mechanical detail. The argument being that it is what you say that matters but not so much how it is said. That of course is not true. Being clear and unambiguous are important aids to any communication in a civilized world.

Similarly, a computer program is a communication between the writer and the reader, whether the reader be that awkward thing: the compiler, another team member, a reviewer or other human soul. Indeed, most communication regarding a program is between two people. Clear and unambiguous syntax is a great help in aiding communication and, as we shall see, avoids a number of common errors.

An important aspect of good syntax design is that it is a worthwhile goal to try to ensure that typical simple typing errors cause the program to become illegal and thus fail to compile, rather than having an unintended meaning. Of course it is hard to prevent the accidental typing of X rather than Y or + rather than * but many structural risks can be prevented. Note incidentally that it is best to avoid short identifiers for just this reason. If we have a financial program about rates and times then using identifiers R and T is risky since we could easily type the wrong identifier by mistake (the letters are next to each other on the keyboard). But if the identifiers are Rate and Time then inadvertently typing Tate or Rime will be caught by the compiler. This applies to any language of course.

Gem # 33: Accessibility Checks (Part I: Ada95)

Author: Ramón Fernández-Marina, AdaCore

Abstract: Ada Gem #33 — The existence of dangling references (pointers to objects that no longer exist) in a program can have catastrophic results. Ada incorporates a set of “accessibility rules” that help the programmer prevent dangling references, making programs more secure.

Let’s get started…

Ada is a block-structured language, which means the programmer can nest blocks of code inside other blocks. At the end of a block, all objects declared inside of it go out of scope, meaning they no longer exist, so the language disallows pointers to objects in blocks with a deeper nesting level.

In order to prevent dangling references, every entity is associated with a number, called its “accessibility level”, according to a Ada’s accessibility rules. When certain references are made to an entity of an access type (Ada’s parlance for pointer), the accessibility level of the entity is checked against the level allowed by the context so that no dangling pointers can occur.

Consider the following example:

```
procedure Static_Check is
   type Global is access all Integer;
   X : Global;

   procedure Init is
      Y : aliased Integer := 0;
   begin
      X := Y’Access; -- Illegal!
   end Init;

begin
   Init;
   …
end Static_Check;
```

The assignment is illegal because when the procedure `Init` finishes, the object `Y` no longer exists, thus making `X` a danging pointer. The compiler will detect this situation and flag the error.

The beauty of the accessibility rules is that most of them can be checked and enforced at compile time, just by using statically known accessibility levels.

However, there are cases when it is not possible to statically determine the accessibility level that an entity will have during program execution. In these cases, the compiler will insert a run-time check to raise an exception if a dangling pointer can be created:

```
procedure Access_Params is
   type Integer_Access is access all Integer;
   Data : Integer_Access;

   procedure Init_Data (Value : access Integer) is
   begin
      Data := Integer_Access (Value);
      -- this conversion performs a dynamic accessibility check
   end;

   X : aliased Integer := 1;

begin
   Init_Data (X’Access); -- This is OK

   declare
      Y : aliased Integer := 2;
   begin
      Init_Data (Y’Access); --  Trouble!
   end;
   --  Y no longer exists!

   Process (Data);
end;
```

In the example above, we cannot know at compile time the accessibility level of the object that will be passed to `Init_Data`, so the compiler inserts a run-time check to make sure that the assignment ‘`Data :=` …’ does not cause a dangling reference — and to raise an exception if it would.

In summary, when it comes to dangling references, Ada makes it very hard for you to shoot yourself in the foot!

Related Source Code

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

Gem #34: Safe and Secure Software : Chapter 2, Safe Typing

Author: John Barnes

Abstract: This week's gem is the second chapter of 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 Gem #30 you can access the contents and bibliography for the entire booklet. We hope you will enjoy the read!

Let's get started…

Safe typing is not about preventing heavy-handed use of the keyboard, although it can detect errors made by typos!

Safe typing is about designing the type structure of the language in order to prevent many common semantic errors. It is often known as strong typing.

Early languages such as Fortran and Algol treated all data as numeric types. Of course, at the end of the day, everything is indeed held in the computer as a numeric of some form, usually as an integer or floating point value and usually encoded using a binary representation. Later languages, starting with Pascal, began to recognize that there was merit in taking a more abstract view of the objects being manipulated. Even if they were ultimately integers, there was much benefit to be gained by treating colors as colors and not as integers by using enumeration types (just called scalar types in Pascal).

Ada take this idea much further as we shall see, but other languages still treat scalar types as just raw numeric types, and miss the critical idea of abstraction, which is to distinguish semantic intent from machine representation. The Ada approach provides more opportunities for detecting programming errors.

Read Chapter 2 in full

Note: All chapters of this booklet will, in time, be available on the Ada 2005 home page.

Related Source Code

Gem #35: bounded buffer package in GNAT hierarchy (Part 1)

Author: Pat Rogers, AdaCore

Abstract: Ada Gem #35 — Ada 95 introduced “protected types” as a fundamental building block for efficient concurrent programming and interrupt handling. In this Gem we examine the use of protected types in the implementation of the classic asynchronous bounded buffer abstraction provided by the GNAT hierarchy of library units. This Gem assumes the reader is somewhat familiar with protected types and will, therefore, explain some, but by no means all, of their semantics.

Let’s get started…

The bounded buffer is a classic concurrent programming component exhibiting asynchronous task interactions. The concept is that of a buffer of a fixed size that is accessed by multiple tasks, some inserting items and some removing them, concurrently and asynchronously. Hence the buffer implementation must be protected against race conditions in which the tasks access the implementation in an interleaved manner and thereby corrupt the representation. In addition to this “mutually exclusive access”, the buffer also requires “condition synchronization”, in which callers are kept waiting until the requested buffer has the necessary state. For example, a task cannot remove an item from a buffer when the buffer is empty. Likewise, an item cannot be put into a buffer when the buffer is full.

Prior to Ada 95, programmers wanting to write portable code had to use the rendezvous to achieve mutual exclusion, with guards to implement the condition synchronization, because no other synchronization mechanism was provided by the language. Although the extended rendezvous has a number of advantages and was a step forward in language design, it has significant overhead when compared to lower-level mechanisms such as semaphores, and is a synchronous mechanism as well. (Ada 80 had a built-in “Semaphore” task type, intended to be implemented efficiently and used as the name suggests, but mixing the higher-level rendezvous with the much lower-level semaphore abstraction was considered poor language design.) In addition, the rendezvous is only available between tasks, meaning that the buffer would have to be implemented as a task too, like the accessing threads. As a result, inserting and removing items would involve expensive task switching, which is the primary source of the comparative inefficiency.

The protected type construct added in Ada 95 addresses this issue directly. Protected types provide efficient mutually exclusive access to encapsulated data, with direct expression of condition synchronization when required. Protected types do not define threads of control, so their use does not involve task switching, and although they do more than simple semaphores, their overhead is comparable.

The GNAT hierarchy of packages includes the generic package GNAT.Bounded_Buffers, providing just the sort of abstraction we have in mind, parameterized for general use. The implementation of the bounded buffer will be that of an array, and we will do assignments of the values held within any given buffer, so the generic formal type representing the values is declared as private, but not limited private or indefinite:

```
generic
   type Element is private;
package GNAT.Bounded_Buffers is
```

Given this generic formal profile, users can instantiate the generic as required. For example, given an appropriate generic actual parameter type named “Job”, we could instantiate it as follows:

```
   package Jobs is new GNAT.Bounded_Buffers (Element => Job);
```

The package declaration contains a pragma Pure so that the generic can be used during library unit elaboration without a potential access-before-elaboration problem. That effect is achieved because Pure units are preelaborated, in addition to other semantics.

Next the package declares the array type used internally in the representation of the bounded buffer type:

```
   type Content is array (Positive range <>) of Element;
```

The array type must be declared outside the protected type, rather than inside in the private part as a hidden implementation artifact. This is an unfortunate holdover from the fact that protected types were originally named “protected records”, with record type semantics: record types cannot declare such things as other types! This limitation was known during the Ada 2005 revision but other revision aspects were more important, so this undesirable restriction remains.

The next declaration in the package is a constant value of type System.Priority:

```
   Default_Ceiling : constant System.Priority :=
      System.Default_Priority;
```

In a real-time application using the Real-Time Systems Annex, protected types are given a “ceiling” priority. The constant declared here is a default for that purpose so that applications not using that Annex can ignore this aspect.

Finally the package declares the protected type itself, with two discriminants:

```
   protected type Bounded_Buffer
      (Capacity : Positive;
       Ceiling  : System.Priority)
   is
      pragma Priority (Ceiling);
```

The first discriminant is the capacity of the instance object, that is, the maximum number of values it can contain. This value will be used in the declaration of a hidden array object of type Content. With this approach, different objects of the one buffer type can have different capacities. The second discriminant represents the ceiling priority value, used in the pragma Priority. This is where the Default_Ceiling constant would be used in non-real-time applications. Note that we cannot use the Default_Ceiling constant as a default discriminant value because the language does not allow some discriminants to have defaults unless all have defaults.

Continuing with our “Jobs” example instantiation, declaration of a bounded buffer specifies these discriminant values:

```
   Buffer : Jobs.Bounded_Buffer (Capacity => 20,
                                 Ceiling => Jobs.Default_Ceiling);
```

In this example we have arbitrarily set the capacity of Buffer to 20. Note that the Bounded_Buffer type is provided directly as a protected type, rather than as a limited private type completed with a protected type. With this approach, clients have full flexibility to do all that protected types allow, such as timed and conditional calls.

Next the protected type declares the visible operations. The two primary operations are Insert and Remove, defined as entries for the sake of the barriers that specify the required condition synchronization. (Only protected entries can have barriers, unlike protected procedures and functions.) The barriers express the “not full” and “not empty” conditions and keep their callers waiting until those conditions hold.

```
      entry Insert (Item : Element);
      entry Remove (Item : out Element);
```

Then three functions are declared. The names “Empty” and “Full” describe the purpose of the first two functions. The third, “Extent”, returns the number of elements currently held in the buffer. It is worth noting that the state of a buffer to which these functions may be applied can change immediately after the call returns.

```
      function Empty return Boolean;
      function Full return Boolean;
      function Extent return Natural;
```

In part two of this Gem we will explore the private part of the protected type, the package body, and the body of the protected type.

Related Source Code

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

Gem #36: Safe and Secure Software : Chapter 3, Safe Pointers

Author: John Barnes

Abstract: This week's gem is the third chapter of 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 Gem #30 you can access the contents and bibliography for the entire booklet. We hope you will enjoy the read!

Let's get started…

Primitive man made a huge leap forward with the discovery of fire. Not only did this allow him to keep warm and cook and thereby expand into more challenging environments but it also enabled the creation of metal tools and thus the bootstrap to an industrial society. But fire is dangerous when misused and can cause tremendous havoc; observe that society has special standing organizations just to deal with fires that are out of control.

Software similarly made a big leap forward in its capabilities when the notion of pointers or references was introduced. But playing with pointers is like playing with fire. Pointers can bring enormous benefits but if misused can bring immediate disaster such as a blue screen, or allow a rampaging program to destroy data, or create the loophole through which a virus can invade.

High integrity software typically limits drastically the use of pointers. The access types of Ada have the semantics of pointers but in addition carry numerous safeguards on their use, which makes them safe in the most demanding safety-critical programs.

1 Read Chapter 3 in full

Note: All chapters of this booklet will, in time, be available on the Ada 2005 home page.

Related Source Code

Gem #37: Bounded Buffer package in GNAT Hierarchy (Part 2)

Author: Pat Rogers, AdaCore

Abstract: Ada Gem #37 — Part 1 of this Gem briefly introduced bounded buffers, protected types, and the declaration of the generic package GNAT.Bounded_Buffers, exporting protected type Bounded_Buffer. In Part 2 of the Gem we examine the private part of Bounded_Buffer and the implementations of the protected entries and functions.

Let's get started…

To recap Part 1, protected types add an efficient building block to the tasking facilities of Ada. Mutual exclusion is automatically provided and condition synchronization is directly expressed via entry barriers, all without the expense of task switching. As such, protected types are a natural implementation approach for a circular bounded buffer used in a concurrent programming context. The complete declaration of the generic package and encapsulated protected type are shown below, including now the private part. (We have omitted the comments in the code since we cover them in the description.)

```
with System;

generic
   type Element is private;
package GNAT.Bounded_Buffers is
   pragma Pure;

   type Content is array (Positive range <>) of Element;

   Default_Ceiling : constant System.Priority :=
      System.Default_Priority;

   protected type Bounded_Buffer
      (Capacity : Positive;
       Ceiling : System.Priority)
   is
      pragma Priority (Ceiling);

      entry Insert (Item : Element);
      entry Remove (Item : out Element);

      function Empty  return Boolean;
      function Full   return Boolean;
      function Extent return Natural;

   private
      Values   : Content (1 .. Capacity);
      Next_In  : Positive := 1;
      Next_Out : Positive := 1;
      Count    : Natural  := 0;
   end Bounded_Buffer;

end GNAT.Bounded_Buffers;
```

The private part of type Bounded_Buffer contains the encapsulated data that can only be accessed via the exported routines. The abstraction is that of a bounded buffer, and it uses the component Values, of the array type Content, to hold the buffer values. The discriminant Capacity determines the actual upper bound on the array component, which is how different Bounded_Buffer objects can have different capacities. As a circular buffer, two array indexes are required, one to indicate where the next inserted value will be placed, and one to indicate where the next removed value will come from. These indexes are components Next_In and Next_Out, respectively. Finally, each Bounded_Buffer object contains a count of the number of elements currently held by the buffer. This counter is used to provide the condition synchronization required to prevent inserting items into a full buffer or removing items from an empty buffer. There are other ways to determine whether the buffer is full or empty but this counter-based approach is clear and is also useful for the function Extent.

The generic package body contains only the body of the protected type, and likewise, the protected body only contains the bodies of the exported entries and functions, so we omit the lines for the declarations of the package and protected type bodies and focus instead on the routines themselves.

The first routine in the protected body is that of the entry Insert. Note the barrier "Count /= Capacity" providing the condition synchronization for the state "not full". Callers to Insert are kept waiting until the barrier is True. As is the case for any protected entry or protected procedure, callers automatically execute with mutually exclusive access to the encapsulated data.

```
entry Insert (Item : Element) when Count /= Capacity is
begin
   Values (Next_In) := Item;
   Next_In := (Next_In mod Capacity) + 1;
   Count := Count + 1;
end Insert;
```

The body of Insert is straightforward. Note that we have the index Next_In wrap around the allowed index values based on the capacity. We could not use a modular type for either index because that would require a static value for the modulus, but we only have the Capacity discriminant to work with.

The body for entry Remove is similar to that of Insert, with expected differences.

```
entry Remove (Item : out Element) when Count > 0 is
begin
   Item := Values (Next_Out);
   Next_Out := (Next_Out mod Capacity) + 1;
   Count := Count - 1;
end Remove;
```

An important point for each entry is the handling of the Count component. When either entry completes, the other entry is examined to see if the barrier has become true. If so, a waiting caller (if any) is allowed to execute and the process repeats itself. Thus, a very simple expression of the condition is provided by the barriers and these are automatically evaluated whenever they could possibly change.

The function bodies are even simpler than the entry bodies. It is worth repeating a point made in Part 1 of this gem, namely that the state of a protected object can change immediately after a call to such a function returns.

```
function Empty return Boolean is
begin
   return Count = 0;
end Empty;

function Full return Boolean is
begin
   return Count = Capacity;
end Full;

function Extent return Natural is
begin
   return Count;
end Extent;
```

Clients can query the capacity of any given Bounded_Buffer object by simply reading the Capacity discriminant. The discriminant cannot be changed, so the effect is much the same as a function.

As Niklaus Wirth once wrote (citing Hoare), once you get the data structures right, the code just writes itself. Admittedly, he didn't use those exact words, but I think you will agree that the data structures used in type Bounded_Buffer, when combined with the semantics of protected types, make the bodies of the routines trivial to write.

Related Source Code

Gem #38: Safe and Secure Software : Chapter 4, Safe Architecture

Author: John Barnes

Abstract: This week's gem is the fourth chapter of 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 Gem #30 you can access the contents and bibliography for the entire booklet. We hope you will enjoy the read!

Let's get started…

When speaking of buildings, a good architecture is one whose design gives the required strength in a natural and unobtrusive manner and thereby provides a safe environment for the people within. An elegant example is the Pantheon in Rome whose spherical shape has enormous strength and provides an uncluttered space. Many ancient cathedrals are not so successful, and need buttresses tacked on the outside to prop up the walls. In 1624, Sir Henry Wooton summed the matter up in his book, The Elements of Architecture, by saying "Well building hath three conditions – commoditie, firmenes & delight". In modern terms, it should work, be strong and be beautiful as well.

A good architecture in a program should similarly provide unobtrusive safety for the detailed workings of the inner parts within a clean framework. It should permit interaction where appropriate and prevent unrelated activities from accidentally interfering with each other. And a good language should enable the writing of programs with a good architecture.

There is perhaps an analogy with the architecture of office spaces. An arrangement where everyone has an individual office can inhibit communication and the flow of ideas. On the other hand, an open plan office often causes problems because noise and other distractions interfere with productivity.

The structure of an Ada program is based primarily around the concept of a package, which groups related entities together and provides a natural framework for hiding implementation details from its clients.

Read Chapter 4 in full

Note: All chapters of this booklet will, in time, be available on the Ada 2005 home page.

Related Source Code

Gem #39: Efficient Stream I/O for Array Types

Author: Pat Rogers, AdaCore

Abstract: Ada Gem #39 — Reading and writing values from/to streams is easy with Ada’s “stream attributes” but for some array types the default attribute implementations could be made more efficient. In this Gem we show how the user can define these more efficient implementations.

Let’s get started…

Ada has the notion of “streams” that are much like those of other languages: sequences of elements comprising values of arbitrary, possibly different, types. Placing a value into a stream is easy using the language-defined “stream attributes”. The programmer simply calls the type-specific attribute routine and specifies the stream and the value. For example, to place an Integer value V into a stream S, one could write the following:

```
Integer’Write (S, V);
```

Strictly speaking, S is not a stream but, rather, an access value designating a stream. The Integer’Write routine will convert the value of V into an array of “stream elements” – essentially an array of storage elements – and then put them into the stream designated by S. Actually, placing the bytes into the stream is accomplished by dynamically dispatching to a procedure specific to the stream representation.

Although this discussion is couched in terms of placing values into streams, you should understand that reading values from streams is very similar to writing them and that the same efficiency issue and solution apply.

For composite types, such as array or record types, each component value is individually written to the stream using the approach described above. Consider an array type “A” specifying Integer as the component type. The default version of A’Write will call Integer’Write for each component. Thus, each Integer value is converted to the array of storage elements and written to the stream. This component-driven behavior is necessary because programmers can define their own versions of the stream attributes, and naturally will expect them to be called even when the types in question are used as component types within enclosing array or record types.

But suppose the array type is structurally just a sequence of contiguous bytes, and the component type does not have a user-defined stream attribute defined. In that case, calling the component-specific attribute for each array component is unnecessary and inefficient.

For example, suppose you are working with Military-Standard 1553B for communicating application values between remote devices. Ultimately, Mil-Std-1553B sends and receives 32-word buffers, where each word is an unsigned 16-bit value. Suppose as well that you want to write and read these buffers to and from streams. We can override the stream attributes so that a whole buffer value is written directly to the stream instead of writing it one buffer component at a time.

The buffer type could be declared as follows:

```
   type Buffer is array (1..32) of Interfaces.Unsigned_16;
```

We can then override the stream attributes for type Buffer.

First we declare the routines:

```
   procedure Read_Buffer
      (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
       Item   : out Buffer);

   procedure Write_Buffer
      (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
       Item   : in Buffer);
```

All such stream attributes have the same formal parameter types, i.e., an access parameter designating the class-wide root stream type defined by the language, and the type to be written to, or read from, that stream.

We then “tie” the routines to the stream attributes for type Buffer, thereby overriding the default versions:

```
   for Buffer'Read  use Read_Buffer;
   for Buffer'Write use Write_Buffer;
```

The language-defined root stream type and array element type are declared in package Ada.Streams:

```
package Ada.Streams is

   type Root_Stream_Type is abstract tagged limited private;

   type Stream_Element is mod 2 ** Standard'Storage_Unit;

   type Stream_Element_Offset is range
     -(2 ** (Standard'Address_Size - 1)) ..
     +(2 ** (Standard'Address_Size - 1)) - 1;

   …

   type Stream_Element_Array is
      array (Stream_Element_Offset range <>) of aliased Stream_Element;

   procedure Read
     (Stream : in out Root_Stream_Type;
      Item   : out Stream_Element_Array;
      Last   : out Stream_Element_Offset)
   is abstract;

   procedure Write
     (Stream : in out Root_Stream_Type;
      Item   : Stream_Element_Array)
   is abstract;

   …
end Ada.Streams;
```

The user-defined Read_Buffer and Write_Buffer routines will call these stream-oriented Read and Write procedures (via dynamic dispatching) once for the entire Buffer array value, instead of calling them once per array component. Both routines are very similar, so we will omit the body of of Read_Buffer for the sake of brevity and show just the implementation of Write_Buffer:

```
   procedure Write_Buffer
      (Stream : not null access Ada.Streams.Root_Stream_Type’Class;
       Item   : in Buffer)
   is
      Item_Size : constant Stream_Element_Offset :=
                     Buffer’Object_Size / Stream_Element’Size;

      type SEA_Pointer is
         access all Stream_Element_Array (1 .. Item_Size);

      function As_SEA_Pointer is
         new Ada.Unchecked_Conversion (System.Address, SEA_Pointer);
   begin
      Ada.Streams.Write (Stream.all, As_SEA_Pointer
                           (Item’Address).all);
   end Write_Buffer;
```

In the above, we cannot simply convert the value of Item, of array type Buffer, to a value of type Stream_Element_Array, so we work with pointers instead. We define an access type designating a Stream_Element_Array that is the exact size, in terms of Stream_Elements, of the incoming Buffer value. Note the use of the Buffer’Object_Size attribute in that computation. That attribute gives us the size of objects of the type Buffer, a wise approach since in general the size of a type may not equal the size of objects of that type. We can then use unchecked conversion to convert the address of the formal parameter Item to this access type. Dereferencing that converted access value (via .all) gives us a value of type Stream_Element_Array that we can pass to the call to Ada.Streams.Write.

Thus we avoid processing each component of type Buffer, instead writing the entire Buffer value at once. That’s a much more efficient approach. As we said earlier, reading values from streams is analogous to writing values to them and only differs in obvious, minor ways. That is true for using the default stream attributes as well as in the implementation of Read_Buffer.

Gem #40: Safe and Secure Software : Chapter 5, Safe Object Oriented Programming

Author: John Barnes

Abstract: This week's gem is the fifth chapter of 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 Gem #30 you can access the contents and bibliography for the entire booklet. We hope you will enjoy the read!

Let's get started…

OOP took programming by storm about twenty years ago. Its supreme merit is said to be its flexibility. But flexibility is somewhat like freedom discussed in the Introduction – the wrong kind of flexibility can be an opportunity that permits dangerous errors to intrude.

The key idea of OOP is that the objects dominate the programming and subprograms (methods) that manipulate objects are properties of objects. The other, older, view sometimes called Function-Oriented (or structured) programming, is that programming is primarily about functional decomposition and that it is the subprograms that dominate program organization, and that objects are merely passive things being manipulated by them.

Both views have their place and fanatical devotion to just a strict object view is often inappropriate.

Ada strikes an excellent balance and enables either approach to be taken according to the needs of the application. Indeed Ada has incorporated the idea of objects right from its inception in 1980 through the concept of packages which encapsulate types and the operations upon them, and tasks that encapsulate independent activities.

Read Chapter 5 in full

Note: All chapters of this booklet will, in time, be available on the Ada 2005 home page.

Related Source Code

Gem #41: Accessibility Checks (Part II: Ada2005)

Author: Ramón Fernández-Marina, AdaCore

Abstract: Ada Gem #41 — Ada 2005 brings a number of improvements concerning access types, aimed at simplifying the programmer's task and adding flexibility to the language. But with greater power comes greater responsibility, so accessibility checks have also been extended to prevent these new features from creating dangling pointers.

Let's get started…

Ada 2005 allows the use of anonymous access types in a more general manner, adding considerable power to the object-oriented programming features of the language. The accessibility rules have been correspondingly augmented to ensure safety by preventing the possibility of dangling references. The new rules have been designed with programming flexibility in mind, as well as to allow the compiler to enforce checks statically.

The accessibility levels in the new contexts for anonymous access types are generally determined by the scope where they are declared. This makes it possible to perform compile-time accessibility checks.

Another rule that allows for static accessibility checks relates to derived types: a type derivation does not create new accessibility level for the derived type, but just takes that of the parent type:

```
procedure Example_1 is
   type Node is record
      N : access Integer;
   end record;
   List : Node

   procedure P is
      type Other_Node is new Node;
   begin
      declare
         L : aliased Integer := 1;
         Data : Other_Node := Other_Node'(N => L'Access);
         --  L'Access is illegal!
      begin
         List := Node (Data);
      end;
   end P;

begin
   P;
end Example_1;
```

In the above example, we don't need to worry about expensive run-time checks on assignment or return of an object of type `Other_Node`; we know it has the same accessibility level as type Node, making the Access attribute illegal. If this were not prevented, after returning from P, List.N would be a dangling reference.

Ada 2005 also allows functions to return objects of anonymous access types. In this case, the accessibility level of the object is statically determined by the scope of the function declaration. Consider the following example:

```
procedure Example_2 is
   type Rec is record
      V : access Integer;
      …
   end record;

   Global : aliased Integer := 1;

   function F1 (X : Boolean) return Rec is
      Local : aliased Integer := 2;

      --  Nested function returns anonymous access values
      --  with different nesting depths

      function F2 (Y : Boolean) return Access Integer is
      begin
         if Y then
            return Global'Access;
         else
            return Local'Access;
         end if;
      end F2;

   begin
      return (V => F2 (X), …); -- Illegal
   end F1;

   Data : Rec;
begin
   Data := F1 (True);
end Example_2;
```

In this example, applying the aforementioned rule, the compiler statically determines that this accessibility level is the scope where F2 is declared, which is deeper than the accessibility level of Rec. So even though the call F1 (True) would provide a valid value for V, the code is illegal. The accessibility restriction is conservative, to keep the rules simple, and so that the compiler is not required to perform data flow analysis to determine legality (not to mention that in general the legality would be undecidable).

The new rules also take into account discriminants of an anonymous access type (which are technically referred to as access discriminants). In Ada 2005, access discriminants are now permitted for nonlimited types. Consequently, it's necessary to disallow defaults for access discriminants of nonlimited types. Thus, the following declaration is illegal:

```
Default : aliased Integer := …
type Rec (D : access Integer := Default’Access) is record
   …
```

This restriction is needed to prevent the discriminant from creating a dangling reference due to an assignment of the record object; it ensures that the object and the discriminant are bound together for their lifetime.

Special care must be taken when types with access discriminants are used with allocators and return statements. The accessibility rules require the compiler to perform static checks when new objects containing access discriminatns are created or returned. Consider the following example:

```
procedure Example_3 is
   type Node (D : access Integer) is record
      V : Integer;
   end record;
   type Ptr is access all Node;

   Global_Value : aliased Integer := 1;
   Other_Data   : Integer := 2;

   procedure P is
      Local : aliased Integer := 3;
      R1 : Ptr;
      R2 : Ptr;
   begin
      R1 := new Node’(D => Global_Value’Access, V => Other_Data);
      --  This is legal

      R2 := new Node’(D => Local_Value’Access, V => Other_Data);
      --  This is illegal
   end P;
begin
   null;
end Example_3;
```

The allocator for `R1` is legal, since the accessibility level of `Global’Access` is the same as the accessibility level of `D`. However the allocator for `R2` is illegal, because the accessibility level of `Local’Access` is deeper than the accessibility level of `D`, and assigning `R2` to an object outside `P` could lead to a dangling reference.

In summary, these rules forbid the creation of an object in a storage pool that contains an access discriminant pointing to some area of memory, be it a part of the stack or some other storage pool, with a shorter lifetime, thus preventing the discriminant from pointing to a nonexistent object.

Related Source Code

Gem #42: Safe and Secure Software : Chapter 6, Safe Object Construction

Author: John Barnes

Abstract: Welcome to the final gem before we take a break for the summer, which is the sixth chapter of 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 Gem #30 you can access the contents and bibliography for the entire booklet.

The gems series will recommence in September. In the meantime, we hope you have a wonderful summer!

Let's get started…

This chapter covers a number of aspects of the control of objects. By objects here we mean both small objects in the sense of simple constants and variables of an elementary type such as Integer and big objects in the sense of Object-Oriented Programming.

Ada provides good control and flexibility in this area. This control is in many cases optional but the good programmer will use the features wherever possible and the good manager will insist upon them being used wherever possible.

Read Chapter 6 in full

Note: All chapters of this booklet will, in time, be available on the Ada 2005 home page.

Related Source Code

Gem #43: Safe and Secure Software : Chapter 7, Safe Memory Management

Author: John Barnes

Abstract: Welcome back the Ada Gems series! We hope you had a great summer.

Gem #43 is the seventh chapter of 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 Gem #30 you can access the contents and bibliography for the entire booklet.

Let's get started…

The memory of the computer provides a vital part of the framework in which the program resides. The integrity of the memory contents is necessary for the health of the program. There is perhaps an analogy with human memory. If the memory is unreliable then the proper functioning of the person is seriously impaired.

There are two main problems with managing computer memory. One is that information can be lost by being improperly overwritten by other information. The other is that the memory itself can become filled and irrecoverable, so that no new information can be stored. This is the problem of memory leaks.

Memory leak is an insidious fault since it often does not show up for a long time. There was an example of a chemical control program that seemed to run flawlessly for several years. It was restarted every three months because of some external constraints (a crane had to be moved which necessitated stopping the plant). But the schedule for the crane changed and the program was then allowed to run for longer – it crashed after four months. There was a memory leak which slowly gnawed away at the free storage.

Read Chapter 7 in full

Note: All chapters of this booklet will, in time, be available on the Ada 2005 home page.

Related Source Code

Gem #44: Accessibility Checks (Part III)

Author: Bob Duff, AdaCore

Abstract: Ada Gem #44 — 'Unchecked_Access can be used to bypass the accessibility rules, and controlled types can be used to rein in this dangerous feature.

Let's get started…

In Gems #1 and #2, we showed how the accessibility rules help prevent dangling pointers, by ensuring that pointers cannot point from longer-lived scopes to shorter-lived ones. But what if you **want** to do that?

In some cases, it is necessary to store a reference to a local object in a global data structure. You can do that by using `'Unchecked_Access` instead of `'Access`. The "`Unchecked`" in the name reminds you that you are bypassing the normal accessibility rules. To prevent dangling pointers, you need to remove the pointer from the global data structure before leaving the scope of the object.

As for any unsafe feature, it is a good idea to encapsulate `'Unchecked_Access`, rather than scattering it all around the program. You can do this using limited controlled types. The idea is that `Initialize` plants a pointer to the object in some global data structure, and `Finalize` removes the pointer just before it becomes a dangling pointer.

Here is an example. Let's assume there are no tasks, and no heap-allocated objects — otherwise, we would need a more complicated data structure, such as a doubly-linked list, with locking. We keep a stack of objects, implemented as a linked list via `Stack_Top` and chained through the `Prev` component. All occurrences of `'Unchecked_Access` are encapsulated in the `Objects` package, and clients of `Objects` (such as `Main`, below at end) can freely declare `Objects`, without worrying about dangling pointers. `Stack_Top` can never dangle, because `Finalize` cleans up, even in the case of exceptions and aborts.

Note that `'Unchecked_Access` is applied to a formal parameter of type `Object`, which is legal because formals of tagged types are defined to be aliased. Note also that `Print_All_Objects` has no visibility on the objects it is printing.

This program prints:

```
Inside Nested:
  That_Object
  This_Object
After Nested returns:
  This_Object
```

Observe that `That_Object` is not printed by the second call to `Print_All_Objects`, because it no longer exists at that time.

```
private with Ada.Finalization;
package Objects is

   type Object (Name : access constant String) is limited private;
   --  The Name is just to illustrate what's going on by
   --  printing it out.

   procedure For_All_Objects
      (Action : not null access procedure (X : Object));
   --  Iterate through all existing Objects in reverse
   --  order of creation,
   --  calling Action for each one.

   procedure Print_All_Objects;
   --  Print out the Names of all Objects in reverse order of creation.

   --  … other operations

private

   use Ada;

   type Object (Name : access constant String) is
      new Finalization.Limited_Controlled with
      record
         --  … other components
         Prev : access Object := null; -- previous Object on the stack
      end record;

   procedure Initialize (X : in out Object);
   procedure Finalize (X : in out Object);

end Objects;

with Ada.Text_IO;
package body Objects is

   Stack_Top : access Object := null;

   procedure Initialize (X : in out Object) is
   begin
      --  Push X onto the stack:
      X.Prev := Stack_Top;
      Stack_Top := X'Unchecked_Access;
   end Initialize;

   procedure Finalize (X : in out Object) is
   begin
      pragma Assert (Stack_Top = X'Unchecked_Access);
      --  Pop X from the stack:
      Stack_Top := X.Prev;
      X.Prev := null;  --  not really necessary, but safe
   end Finalize;

   procedure For_All_Objects
      (Action : not null access procedure (X : Object)) is
      --  Loop through the stack from top to bottom.
      Item : access Object := Stack_Top;
   begin
      while Item /= null loop
         Action (Item.all);
         Item := Item.Prev;
      end loop;
   end For_All_Objects;

   procedure Print_All_Objects is
      --  Iterate through the stack using For_All_Objects, passing
      --  Print_One_Object to print each one.
      procedure Print_One_Object(X : Object) is
      begin
         Text_IO.Put_Line ("  " & X.Name.all);
      end Print_One_Object;
   begin
      For_All_Objects (Print_One_Object'Access);
   end Print_All_Objects;

end Objects;

with Ada.Text_IO; use Ada;
with Objects; use Objects;
procedure Main is

   This_Object : Object (Name => new String'("This_Object"));

   procedure Nested is
      That_Object : Object (Name => new String'("That_Object"));
   begin
      Text_IO.Put_Line ("Inside Nested:");
      Print_All_Objects;
   end Nested;

begin
   Nested;
   Text_IO.Put_Line ("After Nested returns:");
   Print_All_Objects;
end Main;
```

Related Source Code

Gem #45: Safe and Secure Software : Chapter 8, Safe Startup

Author: John Barnes

Abstract: Gem #45 is the eighth chapter of 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 Gem #30 you can access the contents and bibliography for the entire booklet.

Let's get started…

We can carefully write a program so that it behaves properly when running, but it is all to no avail if it will not start properly.

The motor car that will not start is no good even if when going it behaves like a Rolls-Royce.

In the case of a computer program, the key things are to ensure that data is initialized properly and this often means that its various components are initialized in the correct order.

Read Chapter 8 in full

Note: All chapters of this booklet will, in time, be available on the Ada 2005 home page.

Related Source Code

Gem #46: Incompatibilities between Ada 83 and Ada 95

Author: Robert Dewar, AdaCore

Abstract: Ada Gem #46 — Part 1, Unconstrained arrays in generics

Let's get started…

For the most part, Ada 95 is entirely upwards compatible with Ada 83, meaning that correctly written portable Ada 83 code can be compiled with an Ada 95 compiler, and no changes are required.

However, there are some cases of incompatibilities which have to be addressed when dealing with legacy code. You can of course compile in Ada 83 mode using the pragma Ada_83, or the equivalent compiler switch -gnat83 (/83 if you are using VMS). However, this means you can't use any Ada 95 features as you continue development, so it is better to fix these incompatibilities if possible.

In this series of Gems we will address these incompatibilities. Some are really easy, others trickier. In this first installment, we address one of the most common cases.

Consider the following declaration,

```
generic
   type T is private;
package GP is
   …
end GP;
```

In Ada 83 you could instantiate this generic with an unconstrained type, for example:

```
package NP is new GP (String);
```

This is no longer allowed in Ada 95, since this represents a so-called "contract-model violation". If you try to compile the instantiation in Ada 95 mode, you will get an error:

```
8.    package NP is new GP (String);
                            |
      >>> actual for "T" must be a definite subtype
```

String is an indefinite subtype, meaning basically that it is unconstrained. In contrast, a definite subtype means a subtype whose bounds are constrained, so something like

```
subtype S50 is String (1 .. 50);
package NP is new GP (S50);
```

would be legal, but unconstrained String cannot be used.

The contract model for generics says that if a generic compiles error free, then all possible instantiations must compile error free. But if you allow the above instantiation, then whether this is legal or not depends on whether the generic has a declaration of a variable of type T. If it does, then the instantiation is illegal, since you can't have variables of type String without specified bounds.

Let's assume that we don't have any illegal instantiations in the legacy code. That means that, in a case like the above, in fact there are no declarations of variables of type T. In that case, all you have to do is to change the generic as follows:

```
generic
   type T(<>) is private;
package GP is
   …
end GP;
```

The use of (<>) here means that instantiation with an unconstrained subtype is allowed, and consequently the generic will check to ensure that no variables of type T are declared in the template, as you can see from this example:

```
1. generic
2.    type T (<>) is private;
3. package GP is
4.    Var : T;
            |
   >>> unconstrained subtype not allowed (need initialization)

5. end;
```

So, following the Ada 95 rules, the contract-model violation is now avoided. Either the generic allows unconstrained subtypes or it does not, and the generic is properly checked when it is compiled.

In summary, we do have a real incompatibility between Ada 83 and Ada 95 here, but it fixes a real hole in the language, and it is easy to fix up old Ada 83 code to meet the new Ada 95 rules.

Related Source Code

Gem #47: Safe and Secure Software : Chapter 9, Safe Communication

Author: John Barnes

Abstract: Gem #47 is the ninth chapter of 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 Gem #30 you can access the contents and bibliography for the entire booklet.

Let's get started…

A program that doesn't communicate with the outside world in some way is useless although very safe. Such a program might almost be in solitary confinement. A prisoner in solitary confinement is safe in the sense that he cannot hurt other people but he is equally of no use to society either.

So for a program to be useful it must communicate. And if the program is written in a safe way so that it does not have internal dangers, it is largely futile if its communication with the world is unsafe. So safety in communication is important since it is here that the program truly has a useful effect.

It is perhaps worth recalling from the introduction that we characterized the difference between safety-critical and security-critical systems as that the former is where the program must not harm the world whereas the latter is where the world must not harm the program. So communication is the ultimate lynchpin of both safety and security.

Read Chapter 9 in full

Note: All chapters of this booklet will, in time, be available on the Ada 2005 home page.

Gem #48: Extending Interfaces in Ada 2005

Author: Quentin Ochem, AdaCore

Abstract: Ada Gem #48 — Ada 2005 introduced the notion of interfaces for designing object classes. While interfaces are extremely convenient for implementing new hierarchies, they can be difficult to extend once they've started to be used. The addition of a new primitive can break all type derivations, as a type has to implement all abstract primitives inherited from its parents. In this Gem, we'll see two ways to overcome this problem, one OOP-generic, and one specific to Ada 2005.

Let's get started…

The classic OOP way

Let's assume we have the following interface:

```
type Animal is interface;

procedure Eat (Beast : in out Animal) is abstract;
```

All types implementing the Animal interface have to override the Eat operation:

```
type Cat is new Animal with record …

procedure Eat (Beast : in out Cat);
```

Now, after a while, the developer of Animal might feel the need to let animals eat something specific, and would like to add the following operation to the interface:

```
procedure Eat (Beast : in out Animal; Thing : in out A_Thing);
```

Unfortunately, there are hundreds of species of animals implementing this interface, and having to migrate everything will be too painful. Not to mention that most of them don't even need this new way of eating - they're just happy eating some random amount of anonymous food. Extending this interface is just not the way to go - so the extension has to be done separately, in a new interface, such as:

```
type Animal_Extension_1 is interface;

procedure Eat (Beast : in out Animal_Extension_1; Thing : in out
A_Thing) is abstract;
```

So now, Animals that need to rely on this new way of eating will need to be declared, such as:

```
type Cat is new Animal and Animal_Extension_1 with record …
```

Note that it’s even possible to enforce the fact that an extension of Animal has to be an Animal in the first place, by writing:

```
type Animal_Extension_1 is interface and Animal;
```

which will lead to a simpler declaration for type Cat, as there’s no longer a need to extend from two interfaces:

```
type Cat is new Animal_Extension_1 with record …
```

The rest of the code will remain completely untouched thanks to this change. Calls to the new subprogram will require some additional amount of work though, as we’ll first have to check that the type of an Animal that we’re dealing with is indeed a descendant of Animal_Extension_1, and perform a conversion to that interface’s class, before calling the new version of Eat:

```
The_Animal : Animal'Class := ...

if The_Animal in Animal_Extension_1'Class then
   Animal_Extension_1'Class (The_Animal).Eat (Something);
end if;
```

The Ada 2005 Way

Ada 2005 introduces the notion of null procedures. A null procedure is a procedure that is declared using “**is null**” and logically has an empty body. Fortunately, null procedures are allowed in interface definitions - they define the default behavior of such a subprogram as doing nothing. Back to the Animal example, the programmer can declare the interface’s Eat primitive as follows:

```
procedure Eat (Beast : in out Animal; Thing : in out A_Thing) is null;
```

All of our hundreds of kinds of animals will automatically inherit from this procedure, but won’t have to implement it. The addition of this declaration does not break source compatibility with the contract of the Animal interface. Moreover, as no new types are involved, it’s a lot easier to make calls to this subprogram - no more need to check membership or write a type conversion, and we can just write:

```
The_Animal : Animal'Class := ...

The_Animal.Eat (Something);
```

which will execute as a no-op except for animals that have explicitly overridden the primitive.

Related Source Code

Gem #49: Safe and Secure Software: Chapter 10, Safe Concurrency

Author: John Barnes

Abstract: Gem #49 is the tenth chapter of 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 Gem #30 you can access the contents and bibliography for the entire booklet.

Let's get started…

In real life many activities happen in parallel. Human beings do thing in parallel with considerable ease. Females seem to do this better than males – perhaps because they have to rock the baby while cooking the food and keeping the tiger out of the cave. The male typically just concentrates on one thing at a time such as catching that rabbit for dinner – or trying to fin his in-law's cave without asking for directions.

Computers traditionally only do one thing at a time, and the operating system makes it look as if several things are going on in parallel. This is not quite so true these days, since many computers do truly have multiple processors but it still does apply to the vast majority of small computers including those used in process control.

Read Chapter 10 in full

Note: All chapters of this booklet will, in time, be available on the Ada 2005 home page.

Related Source Code

Gem #50: Overload Resolution

Author: Bob Duff, AdaCore

Abstract: Ada Gem #50 — This Gem discusses some language-design issues related to overload resolution.

Let’s get started…

Ada allows overloading of subprograms, which means that two or more subprogram declarations with the same name can be visible at the same place. Here, “name” can refer to operator symbols, like “+”. Ada also allows overloading of various other notations, such as literals and aggregates.

In most languages that support overloading, overload resolution is done “bottom up” — that is, information flows from inner constructs to outer constructs. (As usual, computer folks draw their trees upside-down, with the root at the top.) For example, if we have two procedures Print:

```
procedure Print (S : Sequence);
procedure Print (S : Set);
X : Sequence;
…
Print (X);
```

the type of X determines which Print is meant in the call.

Ada is unusual in that it supports top-down overload resolution as well:

```
function Empty return Sequence;
procedure Print_Sequence (S : Sequence);
function Empty return Set;
procedure Print_Set (S : Set);
…
Print_Sequence (Empty);
```

The type of the formal parameter S of Print_Sequence determines which Empty is meant in the call. In C++, for example, the equivalent of the “Print (X)” example would resolve, but the “Print_Sequence (Empty)” would be illegal, because C++ does not use top-down information.

If we overload things too heavily, we can cause ambiguities:

```
function Empty return Sequence;
procedure Print (S : Sequence);
function Empty return Set;
procedure Print (S : Set);
…
Print (Empty); -- Illegal!
```

The call is ambiguous, and therefore illegal, because there are two possible meanings. One way to resolve the ambiguity is to use a qualified expression to say which type we mean:

```
    Print (Sequence'(Empty));
```

Note that we're now using both bottom-up and top-down overload resolution: Sequence' determines which Empty is meant (top down) and which Print is meant (bottom up). You can qualify an expression, even if it is not ambiguous according to Ada rules — you might want to clarify the type because it might be ambiguous for human readers.

Of course, you could instead resolve the "Print (Empty)" example by modifying the source code so the names are unique, as in the earlier examples. That might well be the best solution, assuming you can modify the relevant sources. Too much overloading can be confusing. How much is "too much" is in part a matter of taste.

Ada really needs to have top-down overload resolution, in order to resolve literals. In some languages, you can tell the type of a literal by looking at it, for example appending "L" (letter el) means "the type of this literal is long int". That sort of kludge won't work in Ada, because we have an open-ended set of integer types:

```
    type Apple_Count is range 0..100;
    procedure Peel (Count : Apple_Count);
    …
    Peel (20);
```

You can't tell by looking at the literal 20 what its type is. The type of formal parameter Count tells us that 20 is an Apple_Count, as opposed to some other type, such as Standard.Long_Integer. [Technically, the type of 20 is universal_integer, which is implicitly converted to Apple_Count -- it's really the result type of that implicit conversion that is at issue. But that's an obscure point -- you won't go _too_ far wrong if you think of the integer literal notation as being overloaded on all integer types.]

Programmers sometimes wonder why the compiler can't resolve something that seems obvious. For example:

```
    type Apple_Count is range 0..100;
    procedure Slice (Count : Apple_Count);
    type Orange_Count is range 0..10_000;
    procedure Slice (Count : Orange_Count);
    …
    Slice (Count => 10_000); -- Illegal!
```

This call is ambiguous, and therefore illegal. But why? Clearly the programmer must have meant the Orange_Count one, because 10_000 is out of range for Apple_Count. And all the relevant expressions happen to be static.

Well, a good rule of thumb in language design (for languages with overloading) is that the overload resolution rules should not be “too smart”. We want this example to be illegal to avoid confusion on the part of programmers reading the code. As usual, a qualified expression fixes it:

```
    Slice (Count => Orange_Count'(10_000));
```

Another example, similar to the literal, is the aggregate. Ada uses a simple rule: the type of an aggregate is determined top down (i.e., from the context in which the aggregate appears). Bottom-up information is not used; that is, the compiler does not look inside the aggregate in order to determine its type.

```
    type Complex is
        record
            Re, Im : Float;
        end record;
    procedure Grind (X : Complex);
    procedure Grind (X : String);
    …
    Grind (X => (Re => 1.0, Im => 1.0)); -- Illegal!
```

There are two Grind procedures visible, so the type of the aggregate could be Complex or String, so it is ambiguous and therefore illegal. The compiler is not required to notice that there is only one type with components Re and Im, of some real type — in fact, the compiler is not _allowed_ to notice that, for overloading purposes.

We can qualify as usual:

```
    Grind (X => Complex'(Re => 1.0, Im => 1.0));
```

Only after resolving that the type of the aggregate is Complex can the compiler look inside and make sure Re and Im make sense.

This not-too-smart rule for aggregates helps prevent confusion on the part of programmers reading the code. It also simplifies the compiler, and makes the overload resolution algorithm reasonably efficient.

How smart is “too smart” is in part a matter of taste. In fact, I would make the Ada rules a little bit less smart, if I were redesigning it from scratch. If we replaced the Grind on String procedure with:

```
    procedure Grind (X : Integer);
```

then the above call would resolve, because the compiler _does_ use the fact that the aggregate must be some sort of aggregate-ish type, like a record or array. I would prefer the call to still be ambiguous in that case, but by and large, Ada gets the rules just about right, so something that is confusingly ambiguous to humans is usually ambiguous by the Ada rules.