# Safe and Secure Software

An Invitation to **Ada 2005**

# 9

## Safe Communication

Courtesy of AdaCore
The GNAT Pro Company

John Barnes

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.

## Representation of data

An important aspect of communication concerns the mapping between the abstract software and the actual hardware. Most languages leave this sort of thing to individual implementations. But Ada gives the user quite specific control over many aspects of data representation.

For example we might decide that we want data in a record to be laid out in a particular manner – perhaps to match that of an existing file structure. Suppose the record is the type Key in the chapter on Safe Object Construction

```
type Key is limited
  record
    Issued: Date;
    Code: Integer;
  end record;
```

where the type Date is

```
type Date is
  record
    Day: Integer range 1 .. 31;
    Month: Integer range 1 .. 12;
    Year: Integer;
  end record;
```

We will assume that we are using a 32-bit machine with four bytes to a word. The day and month easily fit into one byte each and the year needs at most 16 bits so the whole date can be neatly packed into a single word. We can express this by

```
for Date use
  record
    Day at 0 range 0 .. 7;
    Month at 1 range 0 .. 7;
    Year at 2 range 0 .. 15;
  end record;
```

In the case of the type Key, the required structure is simply two words and almost inevitably the implementation will use the representation we require. But we can ensure this by writing

```
for Key use
  record
    Issued at 0 range 0 .. 31;
    Code at 1 range 0 .. 31;
  end record;
```

As another example consider the type Signal of the chapter on Safe Typing. It was

```
type Signal is (Danger, Caution, Clear);
```

Unless we say otherwise, the compiler will encode this type using 0 for Danger, 1 for Caution and 2 for Clear. But in a real application the value of the signal might enter the program encoded as 1 for Danger, 2 for Caution and 4 for Clear. We can instruct the program to use this encoding by writing

```
for Signal use (Danger => 1, Caution => 2, Clear => 4);
```

Furthermore, if the value of The_Signal is autonomously loaded into the program at a particular hardware location as a single byte then we can direct the compiler to ensure that the type is indeed held as such and that the variable is located appropriately by for example

```
for Signal'Size use 8;

for The_Signal'Address use 16#0ACE#;
```

The latter locates the variable at the hexadecimal address 0ACE.

## Validity of data

An important part of all programming is to ensure that data received from the outside world is valid. In most case we can simply program various checks using normal programming techniques. But sometimes this is awkward.

The type Signal is a case in point. We have instructed the compiler to hold the value as an enumeration type with a certain representation. If by some

misfortune a value turns up which does not have a recognized pattern (perhaps two bits are set because of a transient in the external device) then we cannot express a test of that in the normal way because that would take us outside the domain of definition of the type Signal. Instead we can write

```
if not The_Signal'Valid then ...
```

Another approach is to use Unchecked_Conversion. We can read the value in, perhaps as a byte, check it and then if it is acceptable, convert it to the type Signal. First we need the type Byte and the conversion routine

```
type Byte is range 0 .. 255;
for Byte'Size use 8;

function Byte_To_Signal is new Unchecked_Conversion(Byte, Signal);
```

and then

```
Raw_Signal: Byte;
for Raw_Signal'Address use 16#0ACE#;
The_Signal: Signal;
...
case Raw_Signal is
  when 1 | 2 | 4  =>
                                  -- raw value OK, convert it
    The_Signal := Byte_To_Signal(Raw_Signal);
    ...                           -- process valid value
  when others =>
    ...                           -- raw value invalid
    ...                           -- take corrective action
end case;
```

The idea of course is that since the type Byte is simply an integer type we can do normal arithmetic on the value in order to check it. The corrective action might include logging the particular invalid value and so on.

The reader should note a flaw in the above if the value truly is loaded autonomously. Between checking and the conversion, a new value might arrive. So it should be copied into a local variable before being tested and processed.

## Communication with other languages

Many modern large systems are written in a mixture of languages each appropriate to the part of the system concerned. The safety-critical control routines and security-critical input routines might be written in Ada (perhaps in SPARK), the GUI interface might be written in C++, some complex

mathematical analysis might be written in Fortran, some device drivers might be in C and so on.

Many languages have some facilities for interworking with other languages (C++ with C for example) but these are often loosely defined. Ada is perhaps unique in providing well-defined mechanisms within the language standard for interfacing to programs in other languages in general. Ada provides specific facilities for communication with programs and data in C, C++, Fortran and COBOL. In particular, Ada recognizes the representation of types in these other languages such as the arrangement of matrices in Fortran and strings in C so that communication retains type safety.

In a mixed language situation it is thus a good idea to use Ada as the central language so that communication with other languages has the benefit of the type checking provided by the Ada conversion routines.

The general means of communication uses pragmas. Thus suppose we have a C routine called next_byte and we wish to call it from our Ada program as the function Next_Byte. We simply write

```
function Next_Byte return Byte;
pragma Import(C, Next_Byte);
```

The pragma indicates that the calling convention is C and also tells the compiler that there is no Ada body for this function. The pragma can supply a different external name and link name if necessary.

Similarly, if we wish the external C program to call the Ada procedure Action then we can make the name of the Ada procedure available externally by writing

```
procedure Action(D: in Data);
pragma Export(C, Action);
```

Access-to-subprogram types are important for communication with other languages especially when programming interactive systems. For example, suppose we want the procedure Action to be called by the GUI when the mouse is clicked. Suppose that there is a C routine mouse_click that takes the address of the code to be called when the mouse is clicked. We can do this by writing

```
type Response is access procedure (D: in Data);
pragma Convention(C, Response);

procedure Set_Click(P: in Response);
pragma Import(C, Set_Click);

procedure Action(D: in Data);
pragma Convention(C, Action);
...
Set_Click(Action'Access);
```

In this case we have not made the name of the procedure Action visible to the C program because it is called indirectly but we do have to ensure that it uses the C calling convention.

## Streams

A potential difficulty occurs when we transmit values of different types to and from the external world. Output is straightforward because we know the type of the value being transmitted and can use the appropriate format. But input is a problem because typically we do not know what is coming. If a file is uniform and all values are of the same type then we simply have to ensure that we have connected to the correct file. The real difficulty arises when values of different types are involved in the same file. Ada has a number of different filing mechanisms, some are for homogeneous files such as files of all integers or text files; for heterogeneous files we use a stream file.

As a very simple example suppose a file is to have a mixture of values of types Integer, Float and Signal. All types have special attributes 'Read and 'Write for use with streams. On output we simply write

```
S: Stream_Access := Stream(The_File);
...
Integer'Write(S, An_Integer);
Float'Write(S, A_Float);
Signal'Write(S, A_Signal);
```

and this results in a mixture of values of different types on The_File. In the space available we cannot give the full details but S identifies the stream associated with the file.

On input we simply do the reverse

```
Integer'Read(S, An_Integer);
Float'Read(S, A_Float);
Signal'Read(S, A_Signal);
```

If we do the calls in the wrong order then the exception Data_Error will be raised because Ada checks that the item being read is of the correct format.

If we do not know the order in which things are to be read then we need to create a class to cover all the different types involved. In this simple case we might declare a root type

```
type Root is abstract tagged null record;
```

to act as a sort of wrapper and then a series of individual types to encapsulate the real data thus

```
type S_Integer is new Root with
  record
    Value: Integer;
  end record;

type S_Float is new Root with
  record
    Value: Float;
  end record;
...
```

and so on. On output we write

```
Root'Class'Output(S, (Root with An_Integer));
Root'Class'Output(S, (Root with A_Float));
Root'Class'Output(S, (Root with A_Signal));
```

Note that the same procedure is used for all the calls. It first outputs the value of the tag of the specific type and then calls (by dispatching) the appropriate Write attribute.

For input we might write

```
Next_Item: Root'Class := Root'Class'Input(S);
...
Process(Next_Item);
```

The procedure Root'Class'Input reads the tag from the stream and then dispatches to the Read attribute to read the item and finally assigns it as the initial value of the object Next_Item. We can then call some other procedure such as Process by dispatching to do whatever we want. We might assign the value to a particular variable according to its type.

To do this we first declare the abstract procedure for the root type thus

```
procedure Process(X: in Root) is abstract;
```

and then specific procedures such as

```
overriding
procedure Process(X: S_Integer) is
begin
  An_Integer := X.Value;          -- extract value from wrapper
end Process;
```

The procedure Process could of course do anything we like with the value concerned.

This has been a somewhat artificial example. The purpose of it has been to illustrate that Ada can process items of various types in a way that preserves the security of the type model.

## Object factories

We have just seen how the predefined stream mechanism enables us to manipulate values whose types are not known until they are input in some way. The underlying mechanism of reading a tag and then creating an object of the appropriate type is also available to the user in Ada 2005.

Suppose we are manipulating the geometrical objects discussed in the chapter on Safe Object-Oriented Programming. These are of various types such as Circle, Square, Triangle and so on and are all derived from the root type Geometry.Object. We might wish to read values of these objects from a keyboard. For a circle we would expect the values of its two coordinates followed by the radius. For a triangle we would expect the two coordinates plus the values of the three sides and so on. We could declare functions Get_Object to read these values such as

```
function Get_Object return Circle is
begin
  return C: Circle do
    Get(C.X_Coord);  Get(C.Y_Coord);  Get(C.Radius);
  end return;
end Get_Object;
```

The internal calls of Get are calls of predefined procedures to read simple values from the keyboard. The user will have to type some code to indicate which type of object is being supplied. Perhaps the values for a circle could be preceded with the string "Circle"; we will also suppose that we have written a simple function Get_String to read and return such a string.

So now all we have to do is to read the code string, and then call the appropriate procedure Get_Object to create an object of the correct type. The key to this is to use a predefined generic function which, given a tag, returns an object of the corresponding type. In essence it is

```
generic
  type T(<>) is abstract tagged limited private;
  with function Constructor return T is abstract;
function Generic_Dispatching_Constructor(The_Tag: Tag) return T'Class;
```

This generic function has two generic parameters, the first identifies the class of types concerned (such as Geometry.Object from which the types Circle, Square

and Triangle are derived) and a dispatching operation to make objects of the specific types (such as functions Get_Object).

We can now instantiate this generic function to give a constructor function for geometrical objects

```
function Make_Object is
        new Generic_Dispatching_Constructor(Object, Get_Object);
```

A call of Make_Object takes the tag of the specific type concerned, then dispatches to the appropriate function Get_Object and finally returns the value created.

We might decide to declare an access variable to refer to the newly created object thus

```
Object_Ptr: access Object'Class;
```

If the tag value is in a variable Object_Tag (of the type Tag which is defined in the predefined language package Ada.Tags – the generic constructor function is also in this package), then we call Make_Object thus

```
Object_Ptr := new Object'(Make_Object(Object_Tag));
```

and now we have made the new object (perhaps a circle) with the values of its coordinates and radius which were read from the keyboard.

We are not quite finished since we have to convert the string "Circle" which identifies the type concerned into the tag value used for dispatching. A simple way to do this is to write

```
for Circle'External_Tag use "Circle";
for Triangle'External_Tag use "Triangle";
```

and then we can read and convert the external string into the internal tag value by

```
Object_Tag: Tag := Internal_Tag(Get_String);
```

There is of course no need to declare the variable Object_Tag since we can combine the operations into one single statement thus.

```
Object_Ptr := new Object'(Make_Object(Internal_Tag(Get_String));
```

Finally, it should be noted that the above discussion has been slightly simplified. The actual constructor has an auxiliary parameter which we have ignored.

North American Headquarters
104 Fifth Avenue, 15th floor
New York, NY 10011-6901, USA
tel +1 212 620 7300
fax +1 212 807 0162
sales@adacore.com
www.adacore.com

European Headquarters
46 rue d'Amsterdam
75009 Paris, France
tel +33 1 49 70 67 16
fax +33 1 49 70 05 52
sales@adacore.com
www.adacore.com