Safe and Secure Software

An Invitation to Ada 2005

3

Safe Pointers

Courtesy of

AdaCore
The GNAT Pro Company

John Barnes

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 suitable for all but the most demanding safety-critical programs.

References, pointers and addresses

Pointers introduce several opportunities for programming errors such as

- *Type safety violations* – creating an object of one type and then accessing it (through a pointer) as though it were of some other type. Or, more generally, using a pointer to access an object in a manner that is inconsistent with some of the object's semantic properties (for example, assigning to a constant or violating a range constraint).
- *Dangling references* – accessing an object through a pointer after the object has been freed; either a local variable that has gone out of scope, or a dynamically allocated object that has been explicitly freed through some other pointer.
- *Storage leakage* – allocating an object that later becomes inaccessible (and so is "garbage") but which is never freed.

Although the details are different, type safety violations and dangling references may similarly arise if the language allows pointers to subprograms.

Historically, languages have taken different approaches to these problems. Early languages such as Fortran, COBOL and Algol 60 did not have a notion of pointers at the level of the user program. Programs in all languages use addresses for basic operations such as calling a subprogram, but addresses in these languages cannot be directly manipulated by the user.

C (and C++) permit pointers to both heap-allocated and declared (stack-allocated) objects, and also to functions. Although these languages offer some checks, it is basically the programmer's responsibility to use pointers correctly.

For example, since C treats an array as a pointer to its initial element, and allows pointer arithmetic as the equivalent of array indexing, all the necessary low-level ingredients are provided that can get programmers into trouble.

Java and other "pure" object-oriented languages do not expose pointers to the application but rely on pointers and dynamic allocation as the basis of the language semantics. Type checking is preserved, dangling references are prevented (there is no explicit "free"), but to avoid storage leakage the language requires that the implementation provide automatic storage reclamation (garbage collection). This is a reasonable approach for certain kinds of programs. It is still a questionable technology for real-time applications, especially ones with safety-critical or security-critical requirements.

The history of Ada with respect to pointers is interesting. The original version of the language, Ada 83, provided pointers only for dynamic allocation (thus no pointers to declared objects, no pointers to subprograms) and also supplied an explicit free operation known as Unchecked_Deallocation. This preserved type safety, and avoided dangling references caused by pointers to out-of-scope local variables, but introduced the possibility of dangling references through incorrect uses of Unchecked_Deallocation.

The decision to include Unchecked_Deallocation was unavoidable, since the only alternative – requiring implementations to supply Garbage Collection – was not an appropriate option given Ada's intended domain of real-time and high-integrity systems. However, the Ada philosophy is that if a feature defeats checks that are normally performed, then its use must be explicit. And indeed, if we are using Unchecked_Deallocation we need to "with" and then instantiate a generic procedure. (The concepts of a *with clause* and *generic instantiation* are explained in the next chapter.) This somewhat heavyweight syntax both prevents accidental usage and makes our intent clear to whomever needs to read or maintain our code.

Ada 95 extended the Ada 83 mechanism, allowing pointers to declared objects and also to subprograms. Ada 2005 has taken things a bit further – for example, making it easier to pass (pointers to) subprograms as runtime parameters. How these were accomplished without sacrificing safety will be the subject of this chapter.

A final note before going into further detail. Perhaps because pointers and references have a hardware-level connotation, Ada uses the term access types. This enforces the view that values of an access type give access to other objects of some designated type (are like dynamic names for these objects) and should not be thought of as simply machine addresses. Indeed, at the implementation level, the representation of an access value might be different from a physical pointer.

Access types and strong typing

We can declare a variable whose values give access to objects of type T by

```
Ref: access T;
```

If we do not give an initial value then a special value **null** is assumed. X can refer to a normal declared object of type T (which must be marked **aliased**) by

```
Obj: aliased T;
...
Ref := Obj'Access;
```

The analogous C version is:

```
t* ref;

t obj;

ref = &obj;
```

T might be a record type such as

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

so we might have

```
Birthday: aliased Date := (Day => 10, Month => 12, Year => 1815);
AD: access Date := Birthday'Access;
```

and then to retrieve the individual components of the date referred to indirectly by AD we can write for example

```
The_Day: Integer := AD.Day;
```

A variable such as AD can also refer to an object dynamically allocated on the heap (called a storage pool in Ada). We can write

```
AD := new Date'(Day => 27, Month => 11, Year => 1852);
```

(The two dates are those of the birth and death of Ada, Countess of Lovelace after whom the language is named.)

A common application of access types is to create linked lists – we might declare

```
type Cell is
  record
    Next: access Cell;
    Value: Integer;
  end record;
```

and then we can create chains of objects of the type Cell linked together.

Sometimes it is convenient to give a name to an access type

```
type Date_Ptr is access all Date;
```

The "**all**" in the syntax indicates that this named type can refer to both objects on the heap and also to those declared locally on the stack that are marked as **aliased**.

Having to mark objects as **aliased** is a useful safeguard. It alerts the programmer to the fact that the object might be referred to indirectly (good for walkthrough reviews) and it also tells the compiler that the object should not be optimized into a register where it would be difficult to access indirectly.

But the key point is that an access type always identifies the type of the object that its values refer to and strong typing is enforced on assignments, parameter passing, and all other uses. Moreover, an access value always has a legitimate value (which could be **null**). At runtime, whenever we attempt to access an object referred to by an object of the type Date_Ptr, there is a check to ensure that the value is not null – the exception Constraint_Error is raised if this check fails.

We can explicitly state that an access value cannot be null by declaring it as follows

```
WD: not null access Date := Wedding_Day'Access;
```

and then of course it must be given an initial value which is not null. The advantage of a so-called null exclusion is that we are guaranteed that an exception cannot occur when accessing the indirect object.

Finally, note that an access value can denote a component of a composite structure, provided the component type is marked as aliased. For example

```
A: array (1 .. 10) of aliased Integer := (1,2,3,4,5,6,7,8,9,10);
P: access Integer := A(4)'Access;
```

But we cannot perform any incremental operations on P such as P++ or P+1 to make it refer to A(5) as can be done in C. This sort of thing in C is prone to errors since nothing prevents us from pointing beyond either end of the array.

Access types and accessibility

We have just seen that the strong typing of Ada ensures that an access value can never refer to an object of the wrong type. The other requirement our language must satisfy is to ensure that the object referred to cannot cease to exist while access objects still refer to it. This is achieved through the notion of accessibility. Consider

```
package Data is
   type AI is access all Integer;
   Ref1: AI;
end Data;

with Data; use Data;

procedure P is
   K: aliased Integer;
   Ref2: AI;
begin
   Ref2 := K'Access;                  -- illegal

   Ref1 := Ref2;
   ...
end P;
```

This is clearly a very artificial example but illustrates the key points in a small space. The package Data has an access type AI and an object of that type called Ref1. The procedure P declares a local variable K and a local access variable Ref2 also of the type AI and attempts to assign an access to K to the variable Ref2. This is forbidden. It is not so much that the reference to Ref2 is dangerous because both Ref2 and K will cease to exist when we return from a call of the procedure P – the danger is that we might assign the value in Ref2 to the global variable Ref1, which would then contain a reference to K that would be usable after K had ceased to exist.

The basic rule is that the lifetime of the accessed object (such as K) must be at least as long as the lifetime of the specified access type (in this case AI). Here it is not and so the attempt to obtain a pointer to K is illegal.

The rules are phrased in terms of accessibility levels (how deeply nested the declaration of something is) and are mostly static, that is to say checked by the compiler; they incur no cost at run time. But the rules concerning parameters of subprograms that are of anonymous access types are dynamic (that is, require runtime checks). This gives more programming flexibility than would otherwise be possible.

In this short introduction to Ada it is not feasible to go into further details. Suffice it to say that the accessibility rules of Ada prevent dangling references, which can be a source of many subtle and hard-to-diagnose errors in lax languages.

References to subprograms

Ada permits references to procedures and functions to be manipulated in a similar way to references to objects. Both strong typing and accessibility rules apply. For example we can write

```
A_Func: access function (X: Float) return Float;
```

and A_Func is then an object that can only refer to functions that take an argument of the type Float and return an argument of type Float (such as the predefined function Sqrt).

So we can write

```
A_Func := Sqrt'Access;
```

and then

```
X: Float := A_Func(4.0);                    -- indirect call
```

and this will call Sqrt with argument 4.0 and hopefully produce 2.0.

Ada thoroughly checks that the parameters and result always match properly and so we cannot call a function indirectly that has the wrong number or types of parameters. The parameter list and result type constitute what is technically called the *profile* of the function.

Thus consider the predefined function Arctan (the inverse tangent). It takes two parameters

```
function Arctan(Y: Float; X: Float) return Float;
```

and returns the angle θ (in radians) such that tan θ = *Y*/*X*. If we attempt to write

```
A_Func := Arctan'Access;                    -- illegal
Z := A_Func(A);                             -- indirect call prevented
```

then the compiler rejects the code because the profile of Arctan does not match that of A_Func. This is just as well because otherwise the function Arctan would read two items from the runtime stack whereas the indirect call via A_Func placed only one parameter on the stack. This would result in the computation becoming meaningless.

Corresponding checks in Ada occur also across compilation unit boundaries (compilation units are units that can be compiled separately, as explained in the chapter on Safe Architecture). Equivalent mismatches are not prevented in C and this is a common cause of serious errors.

More complex situations arise because a subprogram can have another subprogram as a parameter. Thus we might have a function whose purpose is to solve an equation $Fn(x) = 0$ where the function *Fn* is itself passed as a parameter. Thus

```
function Solve(Trial: Float; Accuracy: Float;
                    Fn: access function (X: Float) return Float)
                                                           return Float;
```

The parameter Trial is the initial guess, the parameter Accuracy is the accuracy required and the third parameter Fn identifies the equation to be solved.

As an example suppose we invest 1000 dollars today and 500 dollars in a year's time: what would the interest rate have to be for the final value two years from now to be exactly 2000 dollars? If the interest rate is x% then the Net Final Value (Nfv) will be given by

$$Nfv(x) = 1000 \times (1 + x/100)^2 + 500 \times (1 + x/100)$$

We can answer the question by declaring the following function, which returns 0.0 when X is such that the net final value is precisely 2000.0.

```
function Nfv_2000 (X: Float) return Float is
  Factor: constant Float := 1.0 + X/100.0;
begin
  return 1000.0 * Factor**2 + 500.0 * Factor – 2000.0;
end Nfv_2000;
```

We can then write:

```
Answer: Float :=
     Solve (Trial => 5.0, Accuracy => 0.01, Fn => Nfv_2000'Access);
```

We are guessing that the answer might be around 5%, we want the answer with 2 decimal figures of accuracy and of course Nfv'Access identifies the problem. The reader is invited to estimate the interest rate – the answer is at the end of this chapter. (Note that terms such as Net Final Value and Net Present Worth are standard terms used by financial professionals.)

The point of this discussion is to emphasize that Ada checks the matching of the parameters of the function parameter as well. Indeed, the nesting of profiles can continue to any degree and Ada matches all levels thoroughly. Many languages give up after one level.

Note that the parameter Fn was actually of an anonymous type. Access to subprogram types can be named or anonymous just like access to object types. They can also have a null exclusion. Thus we should really have written

```
A_Func: not null access function (X: Float) return Float := Sqrt'Access;
```

The advantage of using a null exclusion is that we are guaranteed that the value of A_Func is not null when the function is called indirectly.

If it seems that having to initialize it, perhaps arbitrarily, to Sqrt'Access is distasteful then we could always declare

```
function Default(X: Float) return Float is
begin
  Put("Value not set");  return 0.0;
end Default;
...
A_Func: not null access function (X: Float) return Float := Default'Access;
```

Similarly we should really add **not null** to the profile in Solve thus

```
function Solve(Trial: Float; Accuracy: Float;
     Fn: not null access function (X: Float) return Float) return Float;
```

This ensures that that the actual function corresponding to Fn cannot be null.

Nested subprograms as parameters

We mentioned that accessibility rules also apply to access-to-subprogram values. Suppose we had declared Solve so that the parameter Fn was of a named type and that it and Solve are in some package

```
package Algorithms is
  type A_Function is not null access function (X: Float) return Float;

  function Solve(Trial: Float; Accuracy: Float; Fn: A_Function)
                                                        return Float;
  ...
end Algorithms;
```

Suppose we now decide to express the interest example with the target value passed as a parameter. We might try

```
with Algorithms;  use Algorithms;
function Compute_Interest(Target: Float) return Float is

  function Nfv_T (X: Float) return Float is
    Factor: constant Float := 1.0 + X/100.0;
  begin
```

```
    return 1000.0 * Factor**2 + 500.0 * Factor – Target;
  end Nfv_T;

begin
  return Solve(Trial => 5.0, Accuracy => 0.01, Fn => Nfv_T'Access);
                                                          -- illegal
end Compute_Interest;
```

However, Nfv_T'Access is not allowed as the Fn parameter because it violates the accessibility rules. The trouble is that the function Nfv_T is at an inner level with respect to the type A_Function. (It has to be in order to get hold of the parameter Target.) If Nfv_T'Access had been allowed then we could have assigned this value to a global variable of the type A_Function so that when Compute_Interest had returned we would have still had a reference to Nfv_T even after it had ceased to be accessible. For example

```
Dodgy_Fn: A_Function := Default'Access;          -- a global variable

function Compute_Interest(Target: Float) return Float is

  function Nfv_T(X: Float) return Float is
    ...
  end Nfv_T;

begin
  Dodgy_Fn := Nfv_T'Access;    -- illegal
  ...
end Compute_Interest;
```

and now suppose that after a call of Compute_Interest we execute:

```
Answer := Dodgy_Fn(99.9);        -- would have unpredictable results
```

The call of Dodgy_Fn would attempt to call Nfv_T but that is no longer possible since it is local to Compute_Interest and would attempt to access the parameter Target which no longer exists. The consequences would be unpredictable (a meaningless result, or perhaps an exception would be raised) if Ada did not prevent it. Note that using an anonymous type for the parameter as in the previous section allows passing the nested function as a parameter, but the accessibility checks prevent the assignment to Dogdy_Fn. A runtime check would detect that Nfv_T is more deeply nested than the target access type A_Function, and a Program_Error exception would be raised. So the solution is just to change the package Algorithms thus

```
package Algorithms is
  function Solve(Trial: Float; Accuracy: Float;
                 Fn: not null access function (X: Float) return Float)
                                                        return Float;
end Algorithms;
```

and the original function Compute_Interest is now exactly as before (except that the comment -- *illegal* needs to be removed).

Those of a mischievous mind might suggest that the problem lies with nesting Nfv_T inside Compute_Interest. It would indeed be possible to declare Nfv_T at the outermost level so that no accessibility problem arises, but then the value Target would have to be passed globally through some package – in the style of Fortran Common blocks. We cannot add it as an additional parameter to Nfv_T because the parameters of Nfv_T must match those of Fn. But passing data globally in this way is in fact bad practice. It violates principles of information hiding and abstraction and does not work at all in a multitasking program. Note that the practice of nesting a function within another, where the inner function uses non-local variables (such as Target) is often called a "downward closure".

Downward closures, that is to say passing a pointer to a nested subprogram as a runtime parameter, is a mechanism that is used in several parts of the Ada predefined library, for applications such as iterating over a data structure.

The nesting of subprograms is a natural requirement for these applications because of the need to pass non-local information. This is harder to do in flat languages such as C, C++ and Java. Although type extensions can be used in some languages to model subprogram nesting, this mechanism is less clear and can be a problem for program maintenance.

Finally, some applications need to combine (invoke) algorithms in a nested manner. Thus we might have other useful stuff in the package Algorithms

```
package Algorthms is

  function Solve(Trial: Float; Accuracy: Float;
                 Fn: not null access function (X: Float) return Float)
                                                          return Float;
  function Integrate (Lo, Hi: Float; Accuracy: Float;
                 Fn: not null access function (X: Float) return Float)
                                                          return Float;
  type Vector is array (Positive range <>) of Float;

  procedure Minimize(V: in out Vector; Accuracy: Float;
          Fn: not null access function (V: Vector) return Float);

end Algorithms;
```

The function Integrate is similar to Solve. It computes the definite integral of the function parameter, between the given limits. The procedure Minimize is a little different. It finds those values of the elements of the array V which make the value of the function parameter a minimum. We might have a situation where a cost function is to be minimized and is itself the result of doing an integration and that the values of V are used in the integration (this might seem rather unlikely but the author spent the first few years of his programming life doing just this sort of thing in the chemical industry).

The structure could be

```
with Algorithms;  use Algorithms;
procedure Do_It is

  function Cost(V: Vector) return Float is

    function F(X: Float) return Float is
      Result: Float;
    begin
      ...           -- compute Result using V as well as X
      return Result;
    end F;

  begin
    return Integrate(0.0, 1.0, 0.01, F'Access);
  end Cost;

  A: Vector(1 .. 10);
begin
  ...               -- perhaps read in or set trial values for the vector A

  Minimize(A, 0.01, Cost'Access);

  ...               -- output final values of the vector A.
end Do_It;
```

This all works like a dream in Ada 2005 – just as it did in Algol 60. In other programming languages this is either difficult or requires the use of unsafe constructs with potentially dangling references.

Further examples of the use of access to subprogram types will be found in the chapter on Safe Communication.

Finally, the interest rate that turns the investment of 1000 dollars and 500 dollars into 2000 dollars in two years is about 18.6%. Nice rate if you can get it.