Safe and Secure Software

An Invitation to Ada 2005

2 Safe Typing

Courtesy of AdaCore
The GNAT Pro Company

John Barnes

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.

Using distinct types

Suppose we are monitoring some engineering production and checking for faulty items. We might count the number of good ones and bad ones. We want to stop production if the number of bad ones reaches some limit and perhaps also stop when the number of good ones reaches some other limit. In C or C++ we might have variables

```
int badcount, goodcount;
int b_limit, g_limit;
```

and then perhaps

```
badcount = badcount + 1;
...
if (badcount == b_limit) { ... };
```

and similarly for the good items. Since everything is really an integer, there is nothing to prevent us writing by mistake

```
if (goodcount == b_limit) { ... }
```

where we really should have written `g_limit`. Maybe it was a cut and paste error or a simple typo (g is next to b on a qwerty keyboard). Anyway, since they are integers the compiler will be happy even if we are not.

We could do the same in any language. But Ada gives us the opportunity to be more precise about what we are doing. We can write

```
type Goods is new Integer;
type Bads is new Integer;
```

These declarations introduce new types, which have all the properties of the predefined type Integer (such as operations + and –) and indeed are implemented in the same way, but are nevertheless distinct. We can now write

```
Good_Count, G_Limit: Goods;
Bad_Count, B_Limit: Bads;
```

and now we have quite distinct groups of entities for our manipulation; any accidental mixing will be detected by the compiler and prevent the incorrect program from running. So we can happily write

```
Bad_Count := Bad_Count + 1;

if Bad_Count = B_Limit then
```

but are prevented from writing

```
if Good_Count = B_Limit then              -- illegal
```

since this is a type mismatch.

If we did indeed want to mix the types, perhaps to compare the bad items and good items then we can do a type conversion (known as a cast in other languages) to make the types compatible. Thus we can write

```
if Good_Count = Goods(B_Limit) then
```

Another example might be when computing the percentage of bad objects, where we can convert both counts to the parent type Integer thus

```
100 * Integer(Bad_Count) / (Integer(Bad_Count)+Integer(Good_Count))
```

We can use the same technique to avoid accidental mixing of floating types. Thus when dealing with weights and heights in the chapter on Safe Syntax, rather then

```
My_Height, My_Weight: Float;
```

it would better to write

```
type Inches is new Float;
type Pounds is new Float;

My_Height: Inches := 68.0;
My_Weight: Pounds := 168.0;
```

and then confusion between the two would be detected by the compiler.

Enumerations and integers

In the chapter on Safe Syntax we discussed an example of a railroad crossing which included a test

```
if (the_signal == clear) { ... };
```

```
if The_Signal = Clear then ... end if;
```

in C and Ada respectively. In C the variable the_signal and associated constants such as clear might be declared thus

```
enum signal {
  danger,
  caution,
  clear
};

enum signal the_signal;
```

This convenient notation in fact is simply a shorthand for defining constants danger, caution and clear of type int. And the variable the_signal is also of type int.

As a consequence, nothing can prevent us from assigning a nonsensical value such as 4 to the_signal. In particular, such a nonsensical value might arise from the use of an uninitialized variable. Moreover, suppose other parts of the program are concerned with chemistry and use states anion and cation; nothing would prevent confusion between *cation* and *caution*. We might also be dealing with girls' names such as betty and clare or weapons such as dagger and spear. Nothing prevents confusion between *dagger* and *danger* or *clare* and *clear*.

In Ada we write

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

The_Signal: Signal := Danger;
```

and no confusion can ever arise since an enumeration type in Ada truly is a different type and not a shorthand for an integer type. If we did also have

```
type Ions is (Anion, Cation);
type Names is (Anne, Betty, Clare, ... );
type Weapons is (Arrow, Bow, Dagger, Spear);
```

then the compiler would prevent the compilation of a program that mixed these things up. Moreover the compiler would prevent us from assigning to Clear or Danger since these are literals and this would be as nonsensical as trying to change the value of an integer literal such as 5 by writing

```
5 := 2 + 2;
```

At the machine level the various enumeration types are indeed encoded as integers and we can access the encodings if we really need to, by using the attribute Pos thus

```
Danger_Code: Integer := Signal'Pos(Danger);
```

We can also specify our own encodings, as we shall see in the chapter on Safe Communication.

Incidentally, a very important built-in type in Ada is the type Boolean, which formally has the declaration

```
type Boolean is (False, True);
```

The result of a test such as The_Signal = Clear is of the type Boolean, and there are operations such as **and**, **or**, **not** which operate on Boolean values. It is never possible in Ada to treat an integer value as a Boolean or vice versa. In C it will be recalled, tests yield integer values and zero is treated as false, and nonzero as true. Again we see the danger in

```
if (the_signal == clear)
{
 ...
};
```

Omitting one equals turns the test into an assignment and because C permits an assignment to act as an expression the syntax is acceptable. The error is further compounded since the integer result is treated as a Boolean for the test. So altogether C has several pitfalls illustrated by the one example

- using = for assignment,
- allowing assignments as expressions,
- treating integers as Booleans in conditional expressions.

Most of these flaws have been carried over into C++. None of these issues are present in Ada.

Constraints and subtypes

It is often the case that we know that the value of a certain variable is always going to be within some meaningful range. If so we should say so and thereby make explicit in the program some assumption about the external world. Thus My_Weight could never be negative and would hopefully never exceed 300 pounds. So we can declare

```
My_Weight: Float range 0.0 .. 300.0;
```

or if we had been methodical programmers and had previously declared a floating type Pounds then

```
My_Weight: Pounds range 0.0 .. 300.0;
```

If by mistake the program generates a value outside this range and then attempts to assign it to My_Weight thus

```
My_Weight := Compute_Weight( ... );
```

then the exception Constraint_Error will be raised (or thrown) at run time. We might handle (or catch) this exception in some other part of the program and take remedial action. If we do not, the program will stop and the runtime system will produce an error message indicating where the violation occurred. This all happens automatically – appropriate checks are inserted into the compiled code.

This idea of subranges was first introduced in Pascal and improved in Ada. It is not available in most other languages and we would have to program our own checks all over the place but more likely we wouldn't bother, and any error resulting from violating these bounds would be that much harder to detect.

If we knew that every weight to be dealt with by the program was in a restricted range, then rather than putting a constraint on every variable declaration we can impose it on the type Pounds in the first place.

```
type Pounds is new Float range 0.0 .. 300.0;
```

On the other hand if some weights in the program are unrestricted and it is only the weight of people that are known to lie in a restricted range then we can write

```
type Pounds is new Float;
subtype People_Pounds is Pounds range 0.0 .. 300.0;

My_Weight: People_Pounds;
```

We can also apply constraints and declare subtypes of integer types and enumeration types. Thus when counting good items we would assume that the number was never negative and perhaps that it would never exceed 1000. So we might have

```
type Goods is new Integer range 0 .. 1000;
```

If we just wanted to ensure that it was never negative but did not wish to impose an upper limit then we could write

```
type Goods is new Integer range 0 .. Integer'Last;
```

where Integer'Last gives the upper value of the type Integer. The restriction to positive or nonnegative values is so common that the Ada language provides the following built-in subtypes:

```
subtype Natural is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
```

The type Goods could then be declared as

```
type Goods is new Natural;
```

and this would just impose the lower limit of zero as required.

As an example of a constraint with an enumeration type we might have

```
type Day is (Monday, Tuesday, Wednesday, Thursday, Friday,
                                        Saturday, Sunday);
subtype Weekday is Day range Monday .. Friday;
```

and then we would be prevented from assigning Sunday to a variable of the subtype Weekday.

Inserting constraints as in the above examples may seem to be tiresome but makes the program clearer. Moreover, it enables the compiler and runtime system to verify that the assumptions being expressed by the constraints are indeed correct.

Arrays and constraints

An array is an indexable set of things. As a simple example, suppose we are playing with a pair of dice and wish to record how many throws of each value (from 2 to 12) have been obtained. Since there are 11 possible values, in C we might write

```
int counters[11];

int throw;
```

and this will in fact declare 11 variables referred to as counters[0] to counters[10] and a single integer variable throw.

If we wish to record the result of another throw then we might write:

```
throw = ... ;

counters[throw–2] = counters[throw–2] + 1;
```

Note the need to decrement the throw value by 2, since C arrays are always zero-indexed (that is, have a lower bound of zero). Now suppose the counting mechanism goes wrong (some joker produces a die with 7 spots perhaps or maybe we are generating the throws using a random number generator and we have not programmed it correctly) and a throw of 13 is generated. What happens? The C program does not detect the error but simply computes where counters[11] would be and adds one to that location. Most likely this will be the location of the variable throw itself since it is declared after the array and it will become 14! The program just goes hopelessly wrong.

This is an example of the infamous buffer overflow problem. It is at the heart of many serious and hard-to-detect programming problems. It is ultimately the loophole which permits viruses to attack systems such as Windows. This is discussed further in Chapter 7 on Safe Memory Management.

Now consider the same program in Ada, we can write

```
Counters: array (2 .. 12) of Integer;

Throw: Integer;
```

and then

```
Throw := ... ;

Counters(Throw) := Counters(Throw) + 1;
```

And now if Throw has a rogue value such as 13 then since Ada has runtime checks to ensure that we cannot read or write to a part of an array that does not exist, the exception Constraint_Error is raised and the program is prevented from running wild.

Note that Ada gives control over the lower bound of the array as well as the upper bound. Array indices in Ada do not all start at zero. Lower bounds in real programs are more often one than zero. Specifying the lower bound as 2 in the above example means that the variable throw can be used directly in the index, without the complication of deciding on and subtracting the appropriate offset as in the C version.

The problem with the dice program was not so much that the upper bound of the array was exceeded (that was the symptom) but rather that the value in Throw was out of bounds. We can catch the mistake earlier by declaring a constraint on Throw thus

```
Throw: Integer range 2 .. 12;
```

and now Constraint_Error is raised when we try to assign 13 to Throw. As a consequence the compiler is able to deduce that Throw always has a value appropriate to the range of the array, and no checks will actually be necessary for accessing the array using Throw as an index. Indeed, placing a constraint on variables used for indexing typically reduces the number of runtime checks overall. Incidentally, we can reduce the double appearance of the range 2 .. 12 by writing

```
Throw: Integer range 2 .. 12;
Counters: array (Throw'Range) of Integer;
```

or even more clearly:

```
subtype Dice_Range is Integer range 2 .. 12;
Throw: Dice_Range;
Counters: array (Dice_Range) of Integer;
```

The advantage of only writing the range once is that if we need to change the program (perhaps adding a third die so that the range becomes 3 .. 18) then this only has to be done in one place.

Range checks in Ada are of enormous practical benefit during testing and can be turned off for a production program. Ada compilers are not unique in applying runtime checks in programs. The Whetstone Algol 60 compiler dating from 1962 did it. Ada (like Java) specifies the checks in the language definition itself.

Perhaps it should also be mentioned that we can give names to array types as well. If we had several sets of counter values then it would be better to write

```
type Counter_Array is array (Dice_Range) of Integer;
Counters: Counter_Array;
Old_Counters: Counter_Array;
```

and then if we wanted to copy all the elements of the array Counters into the corresponding elements of the array Old_Counters then we simply write

```
Old_Counters := Counters;
```

Giving names to array types is not possible in many languages. The advantage of naming types is that it introduces *explicit* abstractions, as when counting the good and bad items. By telling the compiler more about what we are doing, we provide it with more opportunities to check that our program makes sense.

Real errors

The title of this section is an example of those nasty puns so hated by the software pioneer Christopher Strachey as mentioned in the Conclusion. This is about accuracy in arithmetic and in particular with real as opposed to integer types.

In floating point arithmetic (using types such as real in Pascal, float in C and Float in Ada) the computation is done with the underlying floating point hardware. Floating point numbers have a relative accuracy. A 32-bit word might allocate 23 bits for the mantissa, one bit for the sign and 8 bits for the exponent. This gives an accuracy of 23 binary digits or about 7 decimal digits.

So a large value such as 123456.7 is accurate to one decimal place, whereas a very small value such as 0.01234567 is accurate to eight decimal places, but in all cases the number of significant digits is always 7. So the accuracy is relative to the magnitude of the number.

Relative accuracy works well most of the time but not always. Consider the representation of an angle giving the bearing of a ship or rocket. Perhaps we would like to hold the accuracy to a second of arc. Remember that there are 60 seconds in a minute, 60 minutes in a degree and 360 degrees in a whole circle.

If we hold the angle as a floating point number

```
float bearing;
```

then the accuracy at 360 degrees will be about 8 seconds which is not good enough, whereas the accuracy at 1 degree will be about 1/45 second which is unnecessary. We could of course hold the value as an integral number of seconds by using an integer type

```
int bearingsecs;
```

This works but it means we have to remember to do our own scaling for input and display purposes.

But the real trouble with floating point is that the accuracy of operations such as addition and subtraction is affected by rounding errors. If we subtract two nearly equal values then we get cancellation errors. And of course certain numbers will not be held exactly. If we have a stepping motor which works in 1/10 degree steps then because 0.1 cannot be held exactly in binary the result of adding 10 steps will not be exactly one degree at all. So even if the accuracy required is quite coarse so that the notional accuracy is more than adequate the cumulative effect of tiny computational errors can be unbounded.

Scaling everything to use integers is acceptable for simple applications but when we have several types held as scaled integers and we have to operate on several together we often get into problems and have to do our own scaling (perhaps even by using raw machine operations such as shifting). This is all prone to errors and difficult to maintain.

Ada is one of the few languages to provide fixed point arithmetic. This does the scaling automatically for us. Thus for the stepping motor we might declare

```
type Angle is delta 0.1 range –360.0 .. 360.0;
for Angle'Small use 0.1;
```

and this will hold the values internally as scaled integers that represent multiples of 0.1 but we can think about them as the abstract values they represent, that is degrees and tenths of degrees. And all arithmetic operations will not suffer from rounding errors.

In summary, Ada has two forms of real arithmetic

- floating point, which provides relative accuracy,
- fixed point, which provides absolute accuracy.

Ada also supplies a specialized form of fixed point for decimal arithmetic, which is the standard model for financial calculations.

The topic of this section is rather specialized but it does illustrate the breadth of facilities in Ada and the care taken to encourage safety in numerical calculations.