Strings in Ada

THE DARK SIDE

Every language has its Achilles heel, so to speak, and I guess with Ada it might be how it handles strings.

First of all, strings in Ada are:

1. NOT references
2. Have a fixed length
3. Are arrays of characters.

WHAT ARE FIXED STRINGS?

A string is an array of characters of a fixed length. For example:

```
s : string(1..10);
r := “DarthVader”;
```

A string is declared implicitly (when it is initialized) or explicitly (with or without initialization):

```
s1 : string := “Yoda”;          -- Implicit len=4. Init.
s2 : string := “Yo”;            -- Implicit len=2. Init.
s3 : string(1..4);              -- Explicit len=4. Uninit.
s4 : string(1..4) := “Yoda”;    -- Explicit len=4. Init.
```

Ada strings are fixed length, meaning that the length of a string NEVER changes. A string variable can only be assigned strings that match its length. Array assignment requires arrays of the same length on both sides. For example:

```
s : string(1..20);
r : string := “DarthVader”;

s := r;    -- compile error
r := s;    -- compile error

r := “Chewbacca!”;  -- this works
r := r & “!”;       -- concatenation causes compile error

s := r & “SithLord!!”;  -- this works
```

Any arrays of the same type can be compared. Strings of different length can be compared. For example:

```
if r = s then
```

String slices can also be compared. For example:

```
if r(1..4) = s(1..4) then
```

Strings have attributes. For example:

```
r : string := “DarthVader”;

put_line(r);            -- “DarthVader”
put_line(r(1..4));      -- “Dart”
put_line(r(5..7));      -- “Vad”
put_line(r(r’first));   -- “V”
put_line(r(r’last));    -- “r”
put(r’length);             -- 10
```

Strings can be sliced to create new strings. For example:

```
s : string(1..5);
r : string := “DarthVader”;

s := r(3..7);           -- “rthVa”
r(6..10) := “Maul!”;    -- “DarthMaul!”
```

NOTE: In Ada.Text_IO, the subprogram **put** is defined for both character and string, however **put_line** is defined ONLY for strings.

A string of length 1, is NOT the same as a character variable. For example:

```
one : string(1..1) := “M”;
one := ‘n’; -- causes a compile error
```

STRINGS IN ADA AREN’T THAT FLEXIBLE

Strings in Ada lack flexibility! In C you may declare a string to have 30 elements, but when you store a word, it is terminated using the end-of-string delimiter ‘\0’. Not so in Ada, so it is not as easy to use.

To get around this you can create a long string and keep track of how many valid characters it contains. For example, using the function **get_line**. The general form of **get_line** is:

```
get_line(str,L);
```

The input string is stored in str, and the number of characters input is stored in L. For example:

```
buf : string(1..40);
len : integer;

get_line(buf, len);
```

If the user enters "**batirex**", then this is stored in **buf**, and the value 7 is stored in **len**.

```
put(buf(1..len));  -- prints out "batirex"
put(len);          -- prints out 7
put(buf);          -- not so happy, avoid this
```

WHAT ABOUT UNBOUNDED STRINGS?

Ada also provides bounded or *unbounded* strings. Unbounded strings are dynamically sized, and are not arrays of characters. They require the use of the packages:

```
ada.strings.unbounded
ada.strings.unbounded_text_io
```

For example:

```
s : unbounded_string;

s := "DarthSidious";
s := "DarthVader";
s := "DarthMinax";
```

Unbounded strings are allocated using heap memory, and are deallocated automatically.

What are the tradeoffs?

- Fixed length strings are time and space efficient, but sometimes inconvenient.
- Unbounded strings are convenient to use, but are time and space inefficient.

2D strings can also be challenging. Knowing the size of the 2D string allows for creating a fixed length 2D string. For example:

```
subtype word is string(1..20);
type maze is array(1..30) of word;
```

This creates a 2D string with 30 words in it of length 20. Sometimes the dimensions are known, so it is best to create a large 2D sting and keep track of the dimensions. For example:

```
type maze is array(1..40) of unbounded_string;
```

This essentially means that there are 40 rows of unbounded strings. The first dimension is fixed, the second dynamic. Given that the dimensions are known, say 20 by 30, the maze could be entered in the following way:

```
m : maze;
s : unbounded_string;

for i in 1..20 loop
    get_line(s);
```

```
        m(i) := s;
    end loop;
```

Accessing the elements in the unbounded string is not that trivial. Trying to print out the 2nd element in the 2nd row using:

```
put(m(2,2));
```

Will result in the following compile error:

```
"too many subscripts in array reference"
```

The proper syntax is:

```
put(element(m(2),2));
```

WHAT ARE THE STRING FUNCTIONS?

There are a series of string functions:

`append` / `&` — concatenate one string to another
`element` — return the character at a particular index
`replace_element` — replace a character at a particular index
`slice` — return a substring (In Ada 2005, also bounded_slice and unbounded_slice)
`replace_slice` / `overwrite` — replace a substring
`insert` — add a string in the midst of the original string
`delete` — remove a string in the midst of the original string
`count` — return the number of occurrences of a substring
`index` — locate a string in the original string
`index_non_blank` — locate the first non-blank character
`head` — return the first character(s) of a string
`tail` — return the last character(s) of a string
`trim` — remove leading or trailing spaces

For example, to replace the 2nd element of the 2nd row with an **X**:

```
replace_element(m(2),2,'X');
```

WHAT ABOUT 2D STRINGS?

Consider the following two ways of creating 2D strings.

The first method is creating an explicit 2D array of characters. For example:

```
type str is array (1..10,1..10) of character;
s1 : str;
```

Each element can now be accessed using the syntax, s1(i,j). For example to set every element to "o":

```
for i in 1..10 loop
    for j in 1..10 loop
        s1(i,j) := 'o';
    end loop;
end loop;
```

The second method is to create an array of strings. For example:

```
type str is array(1..10) of string(1..10);
s1 : str;
```

Elements can no longer be accessed using the form s1(i,j). For example to set every element to "o" can be achieved in the following manner:

```
for i in 1..10 loop
    s1(i) := "oooooooooo";
end loop;
```

Or alternatively using one of the string manipulation procedures found in ada.strings.fixed:

```
for i in 1..10 loop
    for j in 1..10 loop
       overwrite(s1(i),j,"o");
   end loop;
end loop;
```

WHAT ABOUT READING 2D STRINGS FROM A FILE?

There is also the case of reading characters from a file. There are a myriad of ways of doing this. Consider an ASCII file, **maze.txt**, of the form:

```
10
10
oooooooooo
o...o....o
o......ooo
ooooo.oooo
o......ooo
o.oooooo.o
o...oooo.o
ooo......o
ooooooo.oo
oooooooooo
```

First create an appropriate structure to hold the data:

```
type str is array(1..20,1..20) of character;
s1 : str;
dx, dy : integer;
ch : character;
infp : file_type;
```

and use the following syntax to read in the size of the data:

```
open(infp,in_file,"maze.txt");
get(infp,dx);
get(infp,dy);
get(infp,ch);
```

The character variable **ch** reads in the **<return>** after the second 10. Next create a nested loop to read in the character data:

```
for i in 1..dx loop
    for j in 1..dy loop
        get(infp, s1(i,j));
    end loop;
end loop;
```