Surprising behaviour of Fortran (90/95)

Arne Babenhauserheide

August 1, 2013

Contents

1 Introduction

I recently started really learning Fortran (as opposed to just dabbling with existing code until it did what I wanted it to).

Here I document the surprises I found along the way.

As reference: I come from Python, C++ and Lisp.

2 Testing Skelleton

This is a code sample for calculating a base60 value from an integer.

The surprises are taken out of the program and marked with double angle brackets («surprise»). They are documented in the chapter Surprises.

```
program base60
  ! first step: Base60 encode.
  ! reference: http://faruk.akgul.org/blog/tantek-celiks-newbase60-in-python-and-java/
  ! 5000 should be 1PL
  implicit none
  <<declare-function-type-program>>
  <<function-test-calls>>
end program base60
```

Listing 1: *base60-program*: Base60 encoder, program part. For reference see Tantek Çelik's Newbase60 In Python And Java and NewBase60.

```
<<declare-function-type-function>>
  implicit none
  !!! preparation
  <<unchanged-argument>>
  <<parameter>>
  ! work variables
  integer :: n = 0
  integer :: remainder = 0
  ! result
  <<variable-declare-init>>
  ! actual algorithm
  if (number == 0) then
     <<return>>
  end if
  ! calculate the base60 string
  <<variable-reset>>
  n = number ! the input argument: that should be safe to use.
  ! catch number = 0
  do while(n > 0)
     remainder = mod(n, 60)
     n = n/60
     <<indizes-start-at-1>>
     ! write(*,*) number, remainder, n
  end do
<<return-end>>
```

Listing 2: *base60-function*: Base60 encoder, function definition (in the same file)

2.1 Helpers

```
write(*,*) 0, trim(numtosxg(0))
write(*,*) 100000, trim(numtosxg(100000))
write(*,*) 1, trim(numtosxg(1))
write(*,*) 2, trim(numtosxg(2))
write(*,*) 60, trim(numtosxg(60))
write(*,*) 59, trim(numtosxg(59))
```

Listing 3: *function-test-calls*: Function test calls to see the results.

3 Surprises

3.1 I have to declare the return type of a function in the main program and in the function

```
! I have to declare the return type of the function in the main program, too.
character(len=1000) :: numtosxg
```

Listing 4: *declare-function-type-program*: Declare the return type of the function in the program

```
character(len=1000) function numtosxg( number )
```

Listing 5: *declare-function-type-function*: Declare the return type of the function in the function definition again

Alternatively to declaring the function in its header, I can also declare its return type in the declaration block inside the function body:

```
function numtosxg (number)
  character(len=1000) :: numtosxg
end function numtosxg
```

Listing 6: Alternative: Declare the return type of a function inside the function

3.2 Variables in Functions accumulate over several function calls

This even happens, when I initialize the variable when I declare it:

```
character(len=1000) :: res = ""
```

Listing 7: *variable-declare-init*: Declare and initialize the variable.

Due to that I have to begin the algorithm with resetting the required variable.

```
res = "" ! I have to explicitely set res to "", otherwise it
         ! accumulates the prior results!
```

Listing 8: *variable-reset*: reset the variable at each run of the algorithm.

This provides a hint that initialization in a declaration inside a function is purely compile-time.

```
program accumulate
  implicit none
  integer :: acc
  write(*,*) acc(), acc(), acc() ! prints 1 2 3
end program accumulate

integer function acc()
  implicit none
  integer :: ac = 0
  ac = ac + 1
  acc = ac
end function acc
```

Listing 9: Accumulation example. Prints 1 2 3.

```
program accumulate
  implicit none
  integer :: acc
  write(*,*) acc(), acc(), acc() ! prints 1 1 1
end program accumulate

integer function acc()
  implicit none
  integer :: ac
  ac = 0
  ac = ac + 1
  acc = ac
end function acc
```

Listing 10: Safer version (best practices thanks to Alexey)

3.3 parameter vs. intent(in)

Defining a variable as parameter gives a constant, not an unchanged function argument:

```
! constants: marked as parameter: not function parameters, but
! algorithm parameters!
character(len=61), parameter :: base60chars = "0123456789"&
     //"ABCDEFGHJKLMNPQRSTUVWXYZ_abcdefghijkmnopqrstuvwxyz"
```

Listing 11: *parameter*: Constants are marked with the keyword parameter

An argument the function is not allowed to change is defined via intent(in):

```
! input: ensure that this is purely used as input.
! intent is only useful for function arguments.
integer, intent(in) :: number
```

Listing 12: *unchanged-argument*: constant function arguments are marked with intent(in)

3.4 To return values from functions, assign the value to the function itself

This feels surprisingly obvious, but it was surprising to me nontheless.

```
numtosxg = "0"
return
```

Listing 13: *return*: Values can be returned from functions at any time by assigning the function the return value and issuing return to end the function call.

The return statement is only needed when returning within a function. At the end of the function it is implied.

```
  numtosxg = res
end function numtosxg
```

Listing 14: *return-end*: At the end of the function, return is not needed.

3.5 Fortran array indizes start at 1 - and are inclusive

For an algorithm like the example base60, where 0 is identified by the first character of a string, this requires adding 1 to the index.

```
! note that fortran indizes start at 1, not at 0.
res = base60chars(remainder+1:remainder+1)//trim(res)
```

Listing 15: *indizes-start-at-1*: Fortran array-indexing starts at 1.

Also note that the indizes are inclusive. The following actually gets the single letter at index n+1:

```
base60chars(n+1:n+1)
```

Listing 16: Getting a single letter from a character array

In python on the other hand, the second argument of the array is exclusive, so to get the same result you would use [n:n+1]:

```
pythonarray[n:n+1]
```

Listing 17: Python-example for getting a single letter from a list

3.6 I have to trim strings when concatenating

It is necessary to get rid of trailing blanks (whitespace) from the last char to the end of the declared memory space, otherwise there will be huge gaps in combined strings - or you will get missing characters.

```
program test
  character(len=5) :: res
  write(*,*) res ! undefined. In the last run it gave me null-bytes, but
                 ! that is not guaranteed.
  res = "0"
  write(*,*) res ! 0
  res = trim(res)//"a"
  write(*,*) res ! 0a
  res = res//"a"
  write(*,*) res ! 0a: trailing characters are silently removed.
  ! who else expected to see 0aa?
  write(res, '(a, "a")') trim(res) ! without trim, this gives an error!
                                   ! *happy*
  write(*,*) res
end program test
```

Listing 18: Example for string concatenaniong with and without trim.

Hint from Alexey: use trim(adjustl(...)) to get rid of whitespace on the left and the right side of the string. Trim only removes *trailing* blanks.