Writing a commandline tool in Fortran

Arne Babenhauserheide

April 11, 2017

When I finished my Diploma, I thought that Fortran is this horribly unreadable 70th language. I thought it should be removed and that it only lived on due to pure inertia. I thought that its only deeper use were to provide the libraries to make numeric Python faster. Then I actually had to use it. In the beginning I mocked it and didn't understand why anyone would choose Fortran over C. What I saw was mostly Fortran 77. The first thing I wrote was "Fortran surprises" — all the strange things you can stumble over. But bit by bit I realized the similarities with Python. That well-written Fortran actually did not look that different from Python — and much cleaner than C. That it gets stuff done. This year Fortran turns 60 (heise reported in German). And I understand why it is still used. And being an ISO standard it is likely that it will stick with us and keep working many more decades.

Here I want to show you how to write a commandline tool in Fortran.

Contents

1 The first program: Hello world :)

Code to be executed when the program runs is enclosed in `program` and `end program`:

```fortran
program hello
  write (*,*) "Hello World!"
  write (*,*) 'Hello Single Quote!'
end program hello
```

Call this `hello.f90` (`.f` is for the old Fortran 77).

The fastest free compiler is gfortran.

```
gfortran -std=gnu -O3 fortran-hello.f90 -o fortran-hello
./fortran-hello
```

```
Hello World!
Hello Single Quote!
```

That's it. This is your first commandline tool.

2 Reading arguments

Most commandline tools accept arguments. Fortran-developers long resisted this and preferred explicit configuration files, but with 2003 argument parsing entered the standard. The tool for this is `get_command_argument`.

```fortran
program cli
  implicit none ! no implicit declaration: all variables must be declared
  character(1000) :: arg

  call get_command_argument(1, arg) ! result is stored in arg, see
  ! https://gcc.gnu.org/onlinedocs/gfortran/GET_005fCOMMAND_005fARGUMENT.html

  if (len_trim(arg) == 0) then ! no argument given
      write (*,*) "Call me --world!"
  else
      if (trim(arg) == "--world") then
          call get_command_argument(2, arg)
          if (len_trim(arg) == 0) then
              arg = "again!"
          end if
          write (*,*) "Hello ", trim(arg)
          ! trim reduces the fixed-size array to non-blank letters
      end if
  end if
end program
```

```
gfortran -std=gnu -O3 fortran-commandline.f90 -o fortran-helloworld
./fortran-helloworld
```

```
./fortran-helloworld --world World
./fortran-helloworld --world
```

```
Call me --world!
Hello World
Hello again!
```

3 Adding structure with modules

The following restructures the program into modules. If you used any OO tool, you know what this does. `use X, only : a, b, c` gets a, b and c from module x.

Note that you have to declare all variables used in the function at the top of the function.

```
module hello
  implicit none
  character(100),parameter :: prefix = "Hello" ! parameters are constants
  public :: parse_args, prefix
contains
  function parse_args() result ( res )
    implicit none
    character(1000) :: res

    call get_command_argument(1, res)
    if (trim(res) == "--world") then
        call get_command_argument(2, res)
        if (len_trim(res) == 0) then
            res = "again!"
        end if
    end if
  end function parse_args
end module hello

program helloworld
  use hello, only : parse_args, prefix
  implicit none
  character(1000) :: world
  world = parse_args()
  write (*,*) trim(prefix), " ", trim(world)
end program helloworld
```

```
gfortran -std=gnu -O3 fortran-modules.f90 -o fortran-modules
./fortran-modules --world World
```

```
Hello World
```

You can also declare functions as pure (free from side effects). I did not yet check whether the compiler enforces that already, but if it does not do it now, you can be sure that this will be added. Fortran compilers are pretty good at enforcing what you tell them. Do see the fortran surprises for a few hints on how to tell them what you want.

4 Performance considerations

Fortran is fast, really fast. But if you come from C, you need to retrain a bit: The inner loop is the first part of the reference, while with C it is the last part.

The following tests the speed difference when looping over the outer or the inner part. You can get a factor 3-5 difference by having the tight inner loop go over the inner part of the multidimensional array.

Note the L1 cache comments: If you want to get really fast with any language, you cannot ignore the capabilities of your hardware.

Also note that this code works completely naturally on multidimensional arrays.

```
! Thanks to http://infohost.nmt.edu/tcc/help/lang/fortran/time.html
program cheaplooptest
  integer :: i,j,k,s
  integer, parameter :: n=150 ! 50 breaks 32KB L1 cache, 150 breaks 256KB L2 cache
  integer,dimension(n,n,n) :: x, y
  real etime
  real elapsed(2)
  real total1, total2, total3, total4
  x(:,:,:) = 1
  total1 = etime(elapsed)
  print *, "start time ", total1
  ! first index as outer loop
  do s=1,n
     do i=1,n
        do j=1,n
           y(i,j,:) = y(i,j,:) + x(i,j,:)
        end do
     end do
  end do
  total2 = etime(elapsed)
  print *, "time for outer loop", total2 - total1
  ! first index as inner loop is much cheaper (difference depends on n)
  do s=1,n
     do k=1,n
        do j=1,n
```

```
          y(:,j,k) = y(:,j,k) + x(:,j,k)
        end do
      end do
    end do
    total3 = etime(elapsed)
    print *, "time for inner loop", total3-total2
    ! plain copy is slightly slower
    do s=1,n
      y = y + x
    end do
    total4 = etime(elapsed)
    print *, "time for simple loop", total4-total3

end program cheaplooptest
```

```
gfortran -std=gnu -O3 fortran-faster.f90 -o fortran-faster
./fortran-faster
```

```
start time    1.33320000E-02
time for outer loop   18.8833313
time for inner loop  0.750000000
time for simple loop  0.726667404
```

This now seriously looks like Python, but faster by factor 5 to 20.

Just to make it completely clear: The following is how the final test code looks (without the additional looping which make it slow enough to time it).

```
program cleanloop
  integer, parameter :: n=150 ! 50 breaks 32KB L1 cache, 150 breaks 256KB L2 cache
  integer,dimension(n,n,n) :: x, y
  x(:,:,:) = 1
  y = y + x
end program cleanloop
```

That's it. If you want to work with any multidimensional stuff like matrices, that's in most cases exactly what you want. And fast.

5 A full tool: base60

The previous tools were partial solutions. The following is a complete solution, including numerical work (which is where Fortran really shines). And setting the numerical precision. I'm sharing it in total, so you can see everything I needed to do to get it working well.

This implements newbase60 by tantek.

It could be even cleaner, if I could find a way to add complex numbers :)

```
module base60conv
  implicit none ! if you use this here, the module must come before the program in gfortra
  ! constants: marked as parameter: not function parameters, but
  ! algorithm parameters!
  character(len=61), parameter :: base60chars = "0123456789"&
       //"ABCDEFGHJKLMNPQRSTUVWXYZ_abcdefghijkmnopqrstuvwxyz"
  integer, parameter :: longlong = selected_int_kind(32) ! length up to 32 in base10, int(
  integer(longlong), parameter :: sixty = 60
  public :: base60chars, numtosxg, sxgtonum, longlong
  private ! rest is private
contains
  function numtosxg( number ) result ( res )
    implicit none
    !!! preparation
    ! input: ensure that this is purely used as input.
    ! intent is only useful for function arguments.
    integer(longlong), intent(in) :: number
    ! work variables
    integer(longlong) :: n
    integer(longlong) :: remainder
    ! result
    character(len=1000) :: res ! do not initialize variables when
    ! declaring them: That only initializes
    ! at compile time not at every function
    ! call and thus invites nasty errors
    ! which are hard to find.  actual
    ! algorithm
    if (number == 0) then
       res = "0"
       return
    end if
    ! calculate the base60 string

    res = "" ! I have to explicitely set res to "", otherwise it
    ! accumulates the prior results!
    n = number ! the input argument: that should be safe to use.
    ! catch number = 0
    do while(n > 0)
       ! in the first loop, remainder is initialized here.
       remainder = mod(n, sixty)
       n = n/sixty
       ! note that fortran indizes start at 1, not at 0.
```

```
      res = base60chars(remainder+1:remainder+1)//trim(res)
      ! write(*,*) number, remainder, n
    end do
    ! numtosxg = res
  end function numtosxg

  function sxgtonum( base60string ) result ( number )
    implicit none
    ! Turn a base60 string into the equivalent integer (number)
    character(len=*), intent(in) :: base60string
    integer :: i ! running index
    integer :: idx, badchar ! found index of char in string
    integer(longlong) :: number
    ! integer,dimension(len_trim(base60string)) :: numbers ! for later openmp
    badchar = verify(base60string, base60chars)
    if (badchar /= 0) then ! one not
      write(*,"(a,i0,a,a)") "# bad char at position ", badchar, ": ", base60string(badcha
      stop 1 ! with OS-dependent error code 1
    end if

    number = 0
    do i=1, len_trim(base60string)
      number = number * 60
      idx = index(base60chars, base60string(i:i), .FALSE.) ! not backwards
      number = number + (idx-1)
    end do
    ! sxgtonum = number
  end function sxgtonum

end module base60conv

program base60
  ! first step: Base60 encode.
  ! reference: http://faruk.akgul.org/blog/tantek-celiks-newbase60-in-python-and-java/
  ! 5000 should be 1PL
  use base60conv
  implicit none

  integer(longlong) :: tests(14) = (/ 5000, 0, 100000, 1, 2, 60, &
      61, 59, 5, 100000000, 256, 65536, 215000, 16777216 /)
  integer :: i, badchar ! index for the for loop
  integer(longlong) :: n ! the current test to run
  integer(longlong) :: number
  ! program arguments
  character(1000) :: arg
  call get_command_argument(1, arg) ! modern fortran 2003!
  if (len_trim(arg) == 0) then ! run tests
     ! I have to declare the return type of the function in the main program, too.
     ! character(len=1000) :: numtosxg
     ! integer :: sxgtonum
     ! test the functions.
     do i=1,size(tests)
        n = tests(i)
        write(*,"(i12,a,a,i12)") n, " ", trim(numtosxg(n)), sxgtonum(trim(numtosxg(n)))
     end do
  else
     if (trim(arg) == "-r") then
        call get_command_argument(2, arg)
        badchar = verify(arg, " 0123456789")
        if (badchar /= 0) then
           write(*,"(a,i0,a,a)") "# bad char at position ", badchar, ": ", arg(badchar:bad
           stop 1 ! with OS-dependent error code 1
        end if
        read (arg, *) number ! read from arg, write to number
        write (*,*) trim(numtosxg(number))
     else
        write (*,*) sxgtonum(arg)
     end if
  end if
end program base60
```

```
gfortran -std=gnu -O3 fortran-base60.f90 -o fortran-base60
./fortran-base60 P
./fortran-base60 h
./fortran-base60 D
./fortran-base60 PhD
factor $(./fortran-base60 PhD) # yes, it's prime! :)
./fortran-base60 -r 85333
./fortran-base60 "!" || echo $?
echo "^ with error code on invalid input :)"
```

```
 23
 42
 13
 85333
85333: 85333
 PhD
# bad char at position 1: !
1
```

```
^ with error code on invalid input :)
```

6 Conclusion

Fortran done right looks pretty clean. It does have its warts, but not more than all the other languages which are stable enough that the program you write today will still run in 10 years to come. And it is fast. And free.

Why I'm writing this? To save your a few years of lost time I spent adjusting my mistaken distaste for a pretty nice language which got a bad reputation because it once was the language everyone had to learn to get anything done (with sufficient performance). And its code did once look pretty bad, but that's long become ancient history — except for the tools which were so unbelievably good that they are still in use 40 years later.

You can ask "what makes a programming language cool?". One easily overlooked point is: Making your programs still run three decades later. That doesn't look fancy and it doesn't look modern, but it brings a lot of value.

And if you use it where it is strong, Fortran is almost as easy to write as Python, but a lot faster (in terms of CPU requirement for the whole task) with much lower resource consumption (in terms of memory usage and startup time). Should you now ask "what about multiprocessing?", then have a look at OpenMP.