21st Century Fortran Documentation

Release 0.0

Radovan Bast

April 04, 2016

Contents

CHAPTER 1

Introduction

Fortran is not dead - quite the opposite. And it is better than its reputation. I wrote this tutorial after having worked with Fortran (77 and 90+) for over a decade. During that time I have been exposed to many lines of code and I have seen what works and what fails. Good programming practices come from experience. And experience comes from bad programming practices. During the past decade I have written tons of horrible code and made every possible mistake. The result of this learning process and the aim of this tutorial is to give a short introduction into the features of Fortran that you will need and that will work, scale, and remain maintainable and manageable.

Managing code complexity is the key for writing maintainable code. Curiously, many scientists write code without ever worrying about maintainability. But it really is code complexity rather than performance that will decide whether your code will be still alive ten years from now.

CHAPTER 2

Getting started

2.1 Before you start

Fortran is a compiled language so we will need a compiler. It will be useful to install GFortran and CMake.

On Debian-like distributions:

```
$ sudo apt-get install gfortran cmake
```

GFortran and CMake are free. It is no problem to use a Fortran compiler provided by another vendor instead of GFortran (Intel, PGI, Cray, XL).

2.2 Compiling a hello world program

The classic hello world program to get us started:

```
program hello

   implicit none

   print *, 'hello world'

end program
```

Copy-paste it to your editor and save it as `hello.F90`. You can compile it with (example GFortran):

```
$ gfortran hello.F90 -o hello.x
```

Then run the code with:

```
$ ./hello.x

hello world
```

Exciting times. Now we can begin.

2.3 Building portable Fortran code with CMake

Write me ...

2.4 Fixed or free form

Write me ...

CHAPTER 3

Structuring your code

3.1 Subroutines

Write me ...

3.2 Functions

Write me ...

3.3 Modules

Write me ...

3.4 Organizing larger projects

Write me ...

3.5 Passing information within the code

Write me ...

CHAPTER 4

Input and output

4.1 Writing to the screen/stdout

Write me ...

4.2 Writing/reading to/from the disk

Write me ...

CHAPTER 5

Controlling the code flow

5.1 Branching with if/else

Write me ...

5.2 Branching with case

Write me ...

5.3 Loops

Write me ...

CHAPTER 6

Working with arrays

Sooner or later in your code you will need arrays to hold data (floating point numbers, integers, characters, logicals) and you need to allocate space for those. You can allocate this space statically or dynamically. Below we will discuss examples of both.

6.1 Static arrays

Static allocation means that the array size is known at compile time. For instance in the example below we specify an array where we can store 1000*3 double precision floating point coordinates for up to 1000 atoms:

```
integer, parameter :: MAX_NUM_ATOMS = 1000

real(8) :: coordinates(MAX_NUM_ATOMS, 3)
```

Other examples:

```
! a static array holding 200 double precision numbers
real(8) :: array1(200)

! a static 2-dimensional array holding 81 integers
integer :: array2(9, 9)

! a static array holding 401 logicals indexed from 0 to 400
logical :: array3(0:400)
```

There are at least two disadvantages of statically allocated arrays: First, if we need to resize them, we need to recompile the code which is inconvenient. The other disadvantage is that static arrays are always allocated, even if we end up not using them during the calculation.

Therefore the recommendation is to not use static allocations unless the array is small and known to "never" change.

It is also a bad idea to introduce static arrays "temporarily" for testing because you are too lazy to allocate and deallocate dynamically. Very often you will forget to change them later and they remain "temporary" for years or decades until someday somebody writes out of array bounds and this is then no fun.

```
! you:    we will "never" need more than 10000 here
! future: wrong, one day you will
integer :: myarray(10000)
```

6.2 Dynamic arrays

Dynamic is typically better than static, with one exception: if statically allocating program runs out of memory, it crashes immediately. A dynamically allocating program can run ouf of memory late which can be frustrating (see below).

This is how it works:

```
! allocatable 1-dimensional double precision array
real(8), allocatable :: myarray1(:)

! allocatable 2-dimensional integer array
integer, allocatable :: myarray2(:, :)

! here we allocate both
allocate(myarray1(1000))
allocate(myarray2(500, 500))

! in between we do some work ...

! here we deallocate both
deallocate(myarray1)
deallocate(myarray2)
```

6.3 Custom dynamic allocation schemes

In the good old Fortran 77 days dynamic allocation was not possible but it was nevertheless needed. One way out was to statically or dynamically (using another language) allocate a big block of memory at the beginning of the calculation and to manage the memory block during the calculation by subdividing it and to "allocate" and "deallocate" with custom functions. Such a custom dynamic allocation is present in a number of legacy codes. One problem with this is that out of bounds memory access bugs can be difficult to detect because they cannot be detected by the compiler or tools typically designed to detect such bugs. This is because for the compiler and the tools such out of bounds access bugs can appear as regular in-bounds reads and writes because they all can happen within the one big block of memory. The other disadvantage is that code that uses custom dynamic allocation schemes becomes less modular (because the big chunk of memory is often carried around through many levels of routine calls) and less portable (because you cannot reuse a routine which depends on a custom solution that another code may not provide).

6.4 Passing arrays to functions/subroutines

Write me ...

6.5 Friendly advice

If you write a code that allocates possibly a lot of memory late in a possibly long calculation, plan your code for a memory dry-run option so that the code can be run traversing all allocations and deallocations without doing actual computations. This is very helpful in avoiding the otherwise extremely annoying experience of seeing a calculation crash after two weeks of runtime because the code fails to allocate an array late in the calculation.

CHAPTER 7

Bad practices

7.1 Common blocks

7.2 implicit.h

Whenever you see this:

```
#include "implicit.h"
```

or this:

```
implicit real(8) (a-h,o-z)
```

or any `implicit` statement that is not `implicit none`, then run. With the implicit statement you can infer the type (in the above case `real(8)` from the first character of a variable or parameter. In other words `implicit real(8) (a-h,o-z)` means that we do not have to explicitly declare variables/parameters and all variables/parameters starting with i-n will be implicitly integers and all others implicitly double precision numbers.

This may sound like a good idea but is one of the greatest evils of the language. The reason for this is that you will very easily introduce typos which are difficult to detect. This may lead to undefined behavior. The other problem is that `implicit` in combination with common blocks leaves you completely in the blind about where variables are defined and which common blocks are used or unused.

Avoid the `implicit` statement at any cost and always use `implicit none` which forces you to declare all variables. Generations of programmers and your future self will thank you.

7.3 SIXLTR variables

In the good old days six character variables were the norm and a limitation. Today they are not.

Try to guess the meaning of the following variables:

```
NWNABA, DIPDER, TSTINP, FCKDDR, GSQUAD, MSDIDI, SUPMAT
```

Exactly. We have no idea. In the 21st century there is no reason to not use self-explaining variable names. The six character limitation is long gone.

7.4 Fixed-form

7.5 Large static arrays

7.6 Long subroutines

If a subroutine does not fit into your laptop terminal screen, then it is too long. Divide and conquer.

7.7 Functional programming features

7.8 Elemental functions

7.9 Pure functions

CHAPTER 8

Good practices

8.1 Version control

8.2 implicit none

8.3 Implementation hiding

8.4 Module names match file names

Consider you have a module called `some_functionality`:

```
module some_functionality

contains

   ...

end module
```

In this case it is good practice to also call the file either some_functionality.f90 or some_functionality.F90 (for the suffix see next section).

The reason is that if you see a compilation or linking problem in a specific module you know immediately where to find it. It is frustrating to work with projects where the module names do not match file names.

For the same reason it is good practice to use only one module per file instead of packing several modules into one file. The latter is possible but confusing. Be an organized programmer and keep modules separate.

8.5 File suffix

8.6 Explicitly list all data and methods used from a module

Citing from the Zen of Python: "Namespaces are one honking great idea – let's do more of those!"

Namespaces are one honking great idea in Fortran, too.

Therefore the explicit use statement

```
use some_module, only: function1, subroutine1, subroutine2
```

is better than the general statement

```
use some_module
```

for three reasons: 1) the explicit use statement with "only" pollutes the namespace less, 2) the reader of this file can find out from which module functions, subroutines, and variables are imported, and 3) it makes it easy to identify symbols which are not used after a refactoring round.

8.7 Good comments

8.8 Object-oriented programming features

8.9 Private and public methods and data

CHAPTER 9

Parallelization

9.1 MPI

Write me ...

9.2 OpenMP

Write me ...

9.3 CUDA

Write me ...

CHAPTER 10

Performance

CHAPTER 11

Debugging

11.1 Ye olde print statement debugging

11.2 Gdb

11.3 Valgrind

CHAPTER 12

What else is there

12.1 Intrinsic functions and subroutines

12.2 Kinds

12.3 Interoperability with other languages