Gotchas

Variable Initialization Using Initialization Expression

The following code:

```
integer :: a = 5
```

is equivalent to:

```
integer, save :: a = 5
```

and *not* to:

```
integer :: a
a = 5
```

See for example this question.

Floating Point Numbers

Assuming the definitions:

```
integer, parameter :: dp=kind(0.d0)            ! double precision
integer, parameter :: sp=kind(0.0 )            ! single precision
```

Then the following code:

```
real(dp) :: a
a = 1.0
```

is equivalent to:

```
real(dp) :: a
a = 1.0_sp
```

and *not* to:

```
real(dp) :: a
a = 1.0_dp
```

As such, always use the `_dp` suffix as explained in *Floating Point Numbers*. However, the following code:

```
real(dp) :: a
a = 1
```

is equivalent to:

```
real(dp) :: a
a = 1.0_dp
```

And so it is safe to assign integers to floating point numbers without losing any accuracy (but one must be careful about integer division, e.g. `1/2` is equal to `0` and not `0.5`).

C/Fortran Interoperability of Logical

The Fortran standard specifies, that the Fortran type `logical(c_bool)` and C type `bool` are interoperable (as long as `c_bool` returns some positive integer). Unfortunately, for some compilers one must enable this behavior with a specific (non-default) option. In particular, the following options must be used:

Compiler	Extra Compiler Option
gfortran	
ifort	-standard-semantics
PGI	-Munixlogical
Cray	
IBM XL	

Empty *Extra Compiler Option* means that no extra option is needed and things work by default.

If you omit these extra compiler options, then when you pass *logical* to and from Fortran, its value will in general be corrupted when accessed from C. A minimal code example that exemplifies this behavior is at: https://gist.github.com/certik/9744747 When you use these extra compiler options, then everything works as expected and there is no issue.

Conclusion: *always* use these extra compiler options when compiling your Fortran code, unless you have a specific reason not to.