Fortran Best Practices

This page collects a modern canonical way of doing things in Fortran. It is meant to be short, and it is assumed that you already know how to program in other languages (like Python, C/C++, ...) and also know Fortran syntax a bit. Some things in Fortran are obsolete, so this guide only shows the "one correct/canonical modern way" how to do things.

Summary of the language: http://www.cs.umbc.edu/~squire/fortranclass/summary.shtml

Language features are explained at: http://en.wikipedia.org/wiki/Fortran_language_features

The best resource is a recent Fortran standard, for example the Fortran 2003 standard: http://www.j3-fortran.org/doc/year/04/04-007.pdf

Fortran Style Guide

Naming Convention

Ultimately this is a matter of preference. Here is a style guide that we like and that seems to be prevalent in most scientific codes (as well as the Fortran standard library) and you are welcome to follow it.

1. Use lowercase for all Fortran constructs (`do`, `subroutine`, `module`, ...).
2. Follow short mathematical notation for mathematical variables/functions (`Ylm`, `Gamma`, `gamma`, `Enl`, `Rnl`, ...).
3. For other names use all lowercase: try to keep names to one or two syllables; if more are required, use underscores to clarify (`sortpair`, `whitechar`, `meshexp`, `numstrings`, `linspace`, `meshgrid`, `argsort`, `spline`, `spline_interp`, `spline_interpolate`, `stoperr`, `stop_error`, `meshexp_der`).

For example "spline interpolation" can be shortened to `spline_interpolation`, `spline_interpolate`, `spline_interp`, `spline`, but not to `splineint` ("int" could mean integration, integer, etc. — too much ambiguity, even in the clear context of a computational code). This is in contrast to `get_argument()` where `getarg()` is perfectly clean and clear.

The above are general guidelines. In general, choosing the right name certainly depends on the word being truncated as to whether the first syllable is sufficient. Usually it is but clearly not always. Thus some thought should go into step "try to keep names to 2 syllables or less" since it can really affect the indicativeness and simplicity. Simple consistent naming rules are a real help in this regard – for both collaboration and for one's own sanity when going back to some old code you haven't seen in while.

Indentation

Use 4 spaces indentation (this is again a matter of preference as some people prefer 2 or 3 spaces).

Comparison to Other Languages

On the other hand, in most of the rest of the programming world, where the main focus is, in one form or another, on defining and using large sets of complex objects, with tons of properties and behaviors, known only in the code in which they are defined (as opposed to defined by the same notation throughout the literature), it makes more sense to use longer, more descriptive names. The naming conventions one sees used in more general-purpose languages such as C++ and Python, therefore, are perfectly consistent with their more general-purpose missions. But Fortran has a different mission (numerical scientific computing).

Floating Point Numbers

Somewhere create and export a parameter `dp`:

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

and declare floats as:

```
real(dp) :: a, b, c
```

Always write all floating point constants with the `_dp` suffix:

```
1.0_dp, 3.5_dp, 1.34e8_dp
```

and never any other way (see also the gotcha *Floating Point Numbers*).

To print floating point double precision numbers without losing precision, use the `(es23.16)` format (see http://stackoverflow.com/questions/6118231/why-do-i-need-17-significant-digits-and-not-16-to-represent-a-double/).

It is safe to assign integers to floating point numbers without losing accuracy:

```
real(dp) :: a
a = 3
```

In order to impose floating point division (as opposed to integer division `1/2` equal to `0`), one can convert the integer to a floating point number by:

```
real(dp) :: a
a = real(1, dp) / 2       ! 'a' is equal to 0.5_dp
```

Integer Division

Just like in Python 2.x or C, when doing things like `(N-1)/N` where `N` is an integer and you want a floating point division, force the compiler to use floats at the right hand side, for example by:

```
(N - 1.0_dp)/N
```

As long as one of the `/` operands is a float, a floating point division will be used.

Modules and Programs

Only use modules and programs. Always setup a module in the following way:

```
module ode1d
use types, only: dp, pi
use utils, only: stop_error
implicit none
private
public integrate, normalize, parsefunction, get_val, rk4step, eulerstep, &
        rk4step2, get_midpoints, rk4_integrate, rk4_integrate_inward, &
        rk4_integrate_inward2, rk4_integrate3, rk4_integrate4, &
        rk4_integrate_inward4

contains

subroutine get_val(...)
...
end subroutine
...

end module
```

The `implicit none` statement works for the whole module (so you don't need to worry about it). By keeping the `private` empty, all your subroutines/data types will be private to the module by default. Then you export things by putting it into the `public` clause.

Setup programs in the following way:

```
program uranium
use fmesh, only: mesh_exp
use utils, only: stop_error, dp
use dft, only: atom
implicit none

integer, parameter :: Z = 92
real(dp), parameter :: r_min = 8e-9_dp, r_max = 50.0_dp, a = 1e7_dp
...
print *, "I am running"
end program
```

Notice the "explicit imports" (using Python terminology) in the `use` statements. You can also use "implicit imports" like:

```
use fmesh
```

But just like in Python, this should be avoided ("explicit is better than implicit") in most cases.

Arrays

When passing arrays in and out of a subroutine/function, use the following pattern for 1D arrays (it is called *assumed-shape*):

```
subroutine f(r)
real(dp), intent(in) :: r(:)
integer :: n, i
n = size(r)
do i = 1, n
    r(i) = 1.0_dp / i**2
enddo
end subroutine
```

2D arrays:

```
subroutine g(A)
real(dp), intent(in) :: A(:, :)
...
end subroutine
```

and call it like this:

```
real(dp) :: r(5)
call f(r)
```

No array copying is done above. It has the following advantages:

- the shape and size of the array is passed in automatically
- the shape is checked at compile time, the size optionally at runtime
- allows to use strides and all kinds of array arithmetic without actually copying any data.

This should always be your default way of passing arrays in and out of subroutines. However in the following cases one can (or has to) use *explicit-shape* arrays:

- returning an array from a function
- interfacing with C code or legacy Fortran (like Lapack)
- operating on arbitrary shape array with the given function (however there are also other ways to do that, see *Element-wise Operations on Arrays Using Subroutines/Functions* for more information)

To use *explicit-shape* arrays, do:

```
subroutine f(n, r)
integer, intent(in) :: n
real(dp), intent(out) :: r(n)
integer :: i
do i = 1, n
```

```
    r(i) = 1.0_dp / i**2
enddo
end subroutine
```

2D arrays:

```
subroutine g(m, n, A)
integer, intent(in) :: m, n
real(dp), intent(in) :: A(m, n)
...
end subroutine
```

and call it like this:

```
real(dp) :: r(5)
call f(size(r), r)
```

In order to return an array from a function, do:

```
function f(n) result(r)
integer, intent(in) :: n
real(dp) :: r(n)
integer :: i
do i = 1, n
    r(i) = 1.0_dp / i**2
enddo
end function
```

If you want to enforce/check the size of the arrays, put at the beginning of the function:

```
if (size(r) /= 4) stop "Incorrect size of 'r'"
```

To initialize an array, do:

```
integer :: r(5)
r = [1, 2, 3, 4, 5]
```

This syntax is valid since the Fortran 2003 standard and it is the preferred syntax (the old syntax `r = (/ 1, 2, 3, 4, 5 /)` should only be used if you cannot use Fortran 2003).

In order for the array to start with different index than 1, do:

```
subroutine print_eigenvalues(kappa_min, lam)
integer, intent(in) :: kappa_min
real(dp), intent(in) :: lam(kappa_min:)

integer :: kappa
do kappa = kappa_min, ubound(lam, 1)
    print *, kappa, lam(kappa)
end do
end subroutine
```

Multidimensional Arrays

Always access slices as `V(:, 1)`, `V(:, 2)`, or `V(:, :, 1)`, e.g. the colons should be on the left. That way the stride is contiguous and it will be fast. So when you need some slice in your algorithm, always setup the array in a way, so that you call it as above. If you put the colon on the right, it will be slow.

Example:

```
dydx = matmul(C(:, :, i), y) ! fast
dydx = matmul(C(i, :, :), y) ! slow
```

In other words, the “fortran storage order” is: smallest/fastest changing/innermost-loop index first, largest/slowest/outermost-loop index last (“Inner-most are left-most.”). So the elements of a 3D array `A(N1,N2,N3)` are stored, and thus most efficiently accessed, as:

```
do i3 = 1, N3
    do i2 = 1, N2
        do i1 = 1, N1
            A(i1, i2, i3)
        end do
    end do
end do
```

Associated array of vectors would then be most efficiently accessed as:

```
do i3 = 1, N3
    do i2 = 1, N2
        A(:, i2, i3)
    end do
end do
```

And associated set of matrices would be most efficiently accessed as:

```
do i3 = 1, N3
    A(:, :, i3)
end do
```

Storing/accessing as above then accesses always contiguous blocks of memory, directly adjacent to one another; no skips/strides.

When not sure, always rewrite (in your head) the algorithm to use strides, for example the first loop would become:

```
do i3 = 1, N3
    Ai3 = A(:, :, i3)
    do i2 = 1, N2
        Ai2i3 = Ai3(:, i2)
        do i1 = 1, N1
            Ai2i3(i1)
        end do
```

```
    end do
end do
```

the second loop would become:

```
do i3 = 1, N3
    Ai3 = A(:, :, i3)
    do i2 = 1, N2
        Ai3(:, i2)
    end do
end do
```

And then make sure that all the strides are always on the left. Then it will be fast.

Element-wise Operations on Arrays Using Subroutines/Functions

There are three approaches:

- `elemental` subroutines
- *explicit-shape* arrays
- implementing the operation for vectors and write simple wrapper subroutines (that use `reshape` internally) for each array shape

In the first approach, one uses the `elemental` keyword to create a function like this:

```
real(dp) elemental function nroot(n, x) result(y)
integer, intent(in) :: n
real(dp), intent(in) :: x
y = x**(1._dp / n)
end function
```

All arguments (in and out) must be scalars. You can then use this function with arrays of any (compatible) shape, for example:

```
print *, nroot(2, 9._dp)
print *, nroot(2, [1._dp, 4._dp, 9._dp, 10._dp])
print *, nroot(2, reshape([1._dp, 4._dp, 9._dp, 10._dp], [2, 2]))
print *, nroot([2, 3, 4, 5], [1._dp, 4._dp, 9._dp, 10._dp])
print *, nroot([2, 3, 4, 5], 4._dp)
```

The output will be:

```
3.0000000000000000
1.0000000000000000        2.0000000000000000        3.0000000000000000        3.162277
1.0000000000000000        2.0000000000000000        3.0000000000000000        3.162277
1.0000000000000000        1.5874010519681994        1.7320508075688772        1.584893
2.0000000000000000        1.5874010519681994        1.4142135623730951        1.319507
```

In the above, typically `n` is a parameter and `x` is the array of an arbitrary shape, but as you can

see, Fortran does not care as long as the final operation makes sense (if one argument is an array, then the other arguments must be either arrays of the same shape or scalars). If it does not, you will get a compiler error.

The `elemental` keyword implies the `pure` keyword, so the subroutine must be pure (can only use `pure` subroutines and have no side effects).

If the elemental function's algorithm can be made faster using array operations inside, or if for some reason the arguments must be arrays of incompatible shapes, then one should use the other two approaches. One can make `nroot` operate on a vector and write a simple wrappers for other array shapes:

```
function nroot(n, x) result(y)
integer, intent(in) :: n
real(dp), intent(in) :: x(:)
real(dp) :: y(size(x))
y = x**(1._dp / n)
end function

function nroot_0d(n, x) result(y)
integer, intent(in) :: n
real(dp), intent(in) :: x
real(dp) :: y
real(dp) :: tmp(1)
tmp = nroot(n, [x])
y = tmp(1)
end function

function nroot_2d(n, x) result(y)
integer, intent(in) :: n
real(dp), intent(in) :: x(:, :)
real(dp) :: y(size(x, 1), size(x, 2))
y = reshape(nroot(n, reshape(x, [size(x)])), [size(x, 1), size(x, 2)])
end function
```

And use as follows:

```
print *, nroot_0d(2, 9._dp)
print *, nroot(2, [1._dp, 4._dp, 9._dp, 10._dp])
print *, nroot_2d(2, reshape([1._dp, 4._dp, 9._dp, 10._dp], [2, 2]))
```

This will print:

```
3.0000000000000000
1.0000000000000000        2.0000000000000000        3.0000000000000000        3.162277
1.0000000000000000        2.0000000000000000        3.0000000000000000        3.162277
```

Or one can use *explicit-shape* arrays as follows:

```
function nroot(n, k, x) result(y)
integer, intent(in) :: n, k
real(dp), intent(in) :: x(k)
real(dp) :: y(size(x))
y = x**(1._dp / n)
```

```
end function
```

Use as follows:

```
print *, nroot(2, 1, [9._dp])
print *, nroot(2, 4, [1._dp, 4._dp, 9._dp, 10._dp])
print *, nroot(2, 4, reshape([1._dp, 4._dp, 9._dp, 10._dp], [2, 2]))
```

The output is the same as before:

```
3.0000000000000000
1.0000000000000000        2.0000000000000000        3.0000000000000000        3.162277
1.0000000000000000        2.0000000000000000        3.0000000000000000        3.162277
```

Allocatable Arrays

When using allocatable arrays (as opposed to pointers), Fortran manages the memory automatically and it is not possible to create memory leaks.

For example you can allocate it inside a subroutine:

```
subroutine foo(lam)
real(dp), allocatable, intent(out) :: lam
allocate(lam(5))
end subroutine
```

And use somewhere else:

```
real(dp), allocatable :: lam
call foo(lam)
```

When the `lam` symbol goes out of scope, Fortran will deallocate it. If `allocate` is called twice on the same array, Fortran will abort with a runtime error. One can check if `lam` is already allocated and deallocate it if needed (before another allocation):

```
if (allocated(lam)) deallocate(lam)
allocate(lam(10))
```

File Input/Output

To read from a file:

```
integer :: u
open(newunit=u, file="log.txt", status="old")
read(u, *) a, b
close(u)
```

Write to a file:

```
integer :: u
open(newunit=u, file="log.txt", status="replace")
write(u, *) a, b
close(u)
```

To append to an existing file:

```
integer :: u
open(newunit=u, file="log.txt", position="append", status="old")
write(u, *) N, V(N)
close(u)
```

The `newunit` keyword argument to `open` is a Fortran 2008 standard, in older compilers, just replace `open(newunit=u, ...)` by:

```
open(newunit(u), ...)
```

where the `newunit` function is defined by:

```
integer function newunit(unit) result(n)
! returns lowest i/o unit number not in use
integer, intent(out), optional :: unit
logical inuse
integer, parameter :: nmin=10   ! avoid lower numbers which are sometimes reserved
integer, parameter :: nmax=999  ! may be system-dependent
do n = nmin, nmax
    inquire(unit=n, opened=inuse)
    if (.not. inuse) then
        if (present(unit)) unit=n
        return
    end if
end do
call stop_error("newunit ERROR: available unit not found.")
end function
```

Interfacing with C

Write a C wrapper using the `iso_c_binding` module:

```
module fmesh_wrapper

use iso_c_binding, only: c_double, c_int
use fmesh, only: mesh_exp

implicit none

contains

subroutine c_mesh_exp(r_min, r_max, a, N, mesh) bind(c)
real(c_double), intent(in) :: r_min
real(c_double), intent(in) :: r_max
real(c_double), intent(in) :: a
integer(c_int), intent(in) :: N
```

```
real(c_double), intent(out) :: mesh(N)
call mesh_exp(r_min, r_max, a, N, mesh)
end subroutine

! wrap more functions here
! ...

end module
```

You need to declare the length of all arrays (`mesh(N)`) and pass it as a parameter. The Fortran compiler will check that the C and Fortran types match. If it compiles, you can then trust it, and call it from C using the following declaration:

```
void c_mesh_exp(double *r_min, double *r_max, double *a, int *N,
        double *mesh);
```

use it as:

```
int N=5;
double r_min, r_max, a, mesh[N];
c_mesh_exp(&r_min, &r_max, &a, &N, mesh);
```

No matter if you are passing arrays in or out, always allocate them in C first, and you are (in C) responsible for the memory management. Use Fortran to fill (or use) your arrays (that you own in C).

If calling the Fortran `exp_mesh` subroutine from the `c_exp_mesh` subroutine is a problem (CPU efficiency), you can simply implement whatever the routine does directly in the `c_exp_mesh` subroutine. In other words, use the `iso_c_binding` module as a direct way to call Fortran code from C, and you can make it as fast as needed.

Interfacing with Python

Using Cython

To wrap Fortran code in Python, export it to C first (see above) and then write this Cython code:

```
from numpy cimport ndarray
from numpy import empty

cdef extern:
    void c_mesh_exp(double *r_min, double *r_max, double *a, int *N,
            double *mesh)

def mesh_exp(double r_min, double r_max, double a, int N):
    cdef ndarray[double, mode="c"] mesh = empty(N, dtype=double)
    c_mesh_exp(&r_min, &r_max, &a, &N, &mesh[0])
    return mesh
```

The memory is allocated and owned (reference counted) by Python, and a pointer is given to

the Fortran code. Use this approach for both “in” and “out” arrays.

Notice that we didn’t write any C code — we only told fortran to use the C calling convention when producing the ”.o” files, and then we pretended in Cython, that the function is implemented in C, but in fact, it is linked in from Fortran directly. So this is the most direct way of calling Fortran from Python. There is no intermediate step, and no unnecessary processing/wrapping involved.

Using ctypes

Alternatively, you can assign C-callable names to your Fortran routines like this:

```
subroutine mesh_exp(r_min, r_max, a, N, mesh) bind(c, name='mesh_exp')
  real(c_double), intent(in), value :: r_min
  real(c_double), intent(in), value :: r_max
  real(c_double), intent(in), value :: a
  integer(c_int), intent(in), value :: N
  real(c_double), intent(out) :: mesh(N)

  ! ...

end subroutine mesh_exp
```

and use the builtin ctypes Python package to dynamically load shared object files containing your C-callable Fortran routines and call them directly:

```
from ctypes import CDLL, POINTER, c_int, c_double
from numpy import empty

fortran = CDLL('./libmyfortranroutines.so')

mesh = empty(N, dtype="double")
fortran.mesh_exp(c_double(r_min), c_double(r_max), c_double(a), c_int(N),
                 mesh.ctypes.data_as(POINTER(c_double)))
```

Callbacks

There are two ways to implement callbacks to be used like this:

```
subroutine foo(a, k)
use integrals, only: simpson
real(dp) :: a, k
print *, simpson(f, 0._dp, pi)
print *, simpson(f, 0._dp, 2*pi)

contains

real(dp) function f(x) result(y)
real(dp), intent(in) :: x
y = a*sin(k*x)
end function f
```

```
end subroutine foo
```

The traditional approach is to simply declare the `f` dummy variable as a subroutine/function using:

```
module integrals
use types, only: dp
implicit none
private
public simpson

contains

real(dp) function simpson(f, a, b) result(s)
real(dp), intent(in) :: a, b
interface
    real(dp) function f(x)
    use types, only: dp
    implicit none
    real(dp), intent(in) :: x
    end function
end interface
s = (b-a) / 6 * (f(a) + 4*f((a+b)/2) + f(b))
end function

end module
```

The other approach since f2003 is to first define a new type for our callback, and then use `procedure(func)` as the type of the dummy argument:

```
module integrals
use types, only: dp
implicit none
private
public simpson

contains

real(dp) function simpson(f, a, b) result(s)
real(dp), intent(in) :: a, b
interface
    real(dp) function func(x)
    use types, only: dp
    implicit none
    real(dp), intent(in) :: x
    end function
end interface
procedure(func) :: f
s = (b-a) / 6 * (f(a) + 4*f((a+b)/2) + f(b))
end function

end module
```

The new type can also be defined outside of the function (and reused), like:

```
module integrals
```

```
use types, only: dp
implicit none
private
public simpson

interface
    real(dp) function func(x)
    use types, only: dp
    implicit none
    real(dp), intent(in) :: x
    end function
end interface

contains

real(dp) function simpson(f, a, b) result(s)
real(dp), intent(in) :: a, b
procedure(func) :: f
s = (b-a) / 6 * (f(a) + 4*f((a+b)/2) + f(b))
end function

real(dp) function simpson2(f, a, b) result(s)
real(dp), intent(in) :: a, b
procedure(func) :: f
real(dp) :: mid
mid = (a + b)/2
s = simpson(f, a, mid) + simpson(f, mid, b)
end function

end module
```

Type Casting in Callbacks

There are essentially five different ways to do that, each with its own advantages and disadvantages.

The methods I, II and V can be used both in C and Fortran. The methods III and IV are only available in Fortran. The method VI is obsolete and should not be used.

I: Work Arrays

Pass a “work array” or two which are packed with everything needed by the caller and unpacked by the called routine. This is the old way – e.g., how LAPACK does it.

Integrator:

```
module integrals
use types, only: dp
implicit none
private
public simpson

contains
```

```
real(dp) function simpson(f, a, b, data) result(s)
real(dp), intent(in) :: a, b
interface
    real(dp) function func(x, data)
    use types, only: dp
    implicit none
    real(dp), intent(in) :: x
    real(dp), intent(inout) :: data(:)
    end function
end interface
procedure(func) :: f
real(dp), intent(inout) :: data(:)
s = (b-a) / 6 * (f(a, data) + 4*f((a+b)/2, data) + f(b, data))
end function

end module
```

Usage:

```
module test
use types, only: dp
use integrals, only: simpson
implicit none
private
public foo

contains

real(dp) function f(x, data) result(y)
real(dp), intent(in) :: x
real(dp), intent(inout) :: data(:)
real(dp) :: a, k
a = data(1)
k = data(2)
y = a*sin(k*x)
end function

subroutine foo(a, k)
real(dp) :: a, k
real(dp) :: data(2)
data(1) = a
data(2) = k
print *, simpson(f, 0._dp, pi, data)
print *, simpson(f, 0._dp, 2*pi, data)
end subroutine

end module
```

II: General Structure

Define general structure or two which encompass the variations you actually need (or are even remotely likely to need going forward). This single structure type or two can then change if needed as future needs/ideas permit but won't likely need to change from passing, say, real numbers to, say, and instantiation of a text editor.

Integrator:

```
module integrals
use types, only: dp
implicit none
private
public simpson, context

type context
    ! This would be adjusted according to the problem to be solved.
    ! For example:
    real(dp) :: a, b, c, d
    integer :: i, j, k, l
    real(dp), pointer :: x(:), y(:)
    integer, pointer :: z(:)
end type

contains

real(dp) function simpson(f, a, b, data) result(s)
real(dp), intent(in) :: a, b
interface
    real(dp) function func(x, data)
    use types, only: dp
    implicit none
    real(dp), intent(in) :: x
    type(context), intent(inout) :: data
    end function
end interface
procedure(func) :: f
type(context), intent(inout) :: data
s = (b-a) / 6 * (f(a, data) + 4*f((a+b)/2, data) + f(b, data))
end function

end module
```

Usage:

```
module test
use types, only: dp
use integrals, only: simpson, context
implicit none
private
public foo

contains

real(dp) function f(x, data) result(y)
real(dp), intent(in) :: x
type(context), intent(inout) :: data
real(dp) :: a, k
a = data%a
k = data%b
y = a*sin(k*x)
end function

subroutine foo(a, k)
real(dp) :: a, k
type(context) :: data
data%a = a
data%b = k
```

```
print *, simpson(f, 0._dp, pi, data)
print *, simpson(f, 0._dp, 2*pi, data)
end subroutine

end module
```

There is only so much flexibility really needed. For example, you could define two structure types for this purpose, one for Schroedinger and one for Dirac. Each would then be sufficiently general and contain all the needed pieces with all the right labels.

Point is: it needn't be "one abstract type to encompass all" or bust. There are natural and viable options between "all" and "none".

III: Private Module Variables

Hide the variable arguments completely by passing in module variables.

Integrator:

```
module integrals
use types, only: dp
implicit none
private
public simpson

contains

real(dp) function simpson(f, a, b) result(s)
real(dp), intent(in) :: a, b
interface
    real(dp) function func(x)
    use types, only: dp
    implicit none
    real(dp), intent(in) :: x
    end function
end interface
procedure(func) :: f
s = (b-a) / 6 * (f(a) + 4*f((a+b)/2) + f(b))
end function

end module
```

Usage:

```
module test
use types, only: dp
use integrals, only: simpson
implicit none
private
public foo

real(dp) :: global_a, global_k

contains
```

```
real(dp) function f(x) result(y)
real(dp), intent(in) :: x
y = global_a*sin(global_k*x)
end function

subroutine foo(a, k)
real(dp) :: a, k
global_a = a
global_k = k
print *, simpson(f, 0._dp, pi)
print *, simpson(f, 0._dp, 2*pi)
end subroutine

end module
```

However it is best to avoid such global variables – even though really just semi-global – if possible. But sometimes it may be the simplest cleanest way. However, with a bit of thought, usually there is a better, safer, more explicit way along the lines of II or IV.

IV: Nested functions

Integrator:

```
module integrals
use types, only: dp
implicit none
private
public simpson

contains

real(dp) function simpson(f, a, b) result(s)
real(dp), intent(in) :: a, b
interface
    real(dp) function func(x)
    use types, only: dp
    implicit none
    real(dp), intent(in) :: x
    end function
end interface
procedure(func) :: f
s = (b-a) / 6 * (f(a) + 4*f((a+b)/2) + f(b))
end function

end module
```

Usage:

```
subroutine foo(a, k)
use integrals, only: simpson
real(dp) :: a, k
print *, simpson(f, 0._dp, pi)
print *, simpson(f, 0._dp, 2*pi)

contains
```

```
real(dp) function f(x) result(y)
real(dp), intent(in) :: x
y = a*sin(k*x)
end function f

end subroutine foo
```

V: Using type(c_ptr) Pointer

In C, one would use the `void *` pointer. In Fortran, one can use `type(c_ptr)` for exactly the same purpose.

Integrator:

```
module integrals
use types, only: dp
use iso_c_binding, only: c_ptr
implicit none
private
public simpson

contains

real(dp) function simpson(f, a, b, data) result(s)
real(dp), intent(in) :: a, b
interface
    real(dp) function func(x, data)
    use types, only: dp
    implicit none
    real(dp), intent(in) :: x
    type(c_ptr), intent(in) :: data
    end function
end interface
procedure(func) :: f
type(c_ptr), intent(in) :: data
s = (b-a) / 6 * (f(a, data) + 4*f((a+b)/2, data) + f(b, data))
end function

end module
```

Usage:

```
module test
use types, only: dp
use integrals, only: simpson
use iso_c_binding, only: c_ptr, c_loc, c_f_pointer
implicit none
private
public foo

type f_data
    ! Only contains data that we need for our particular callback.
    real(dp) :: a, k
end type

contains
```

```
real(dp) function f(x, data) result(y)
real(dp), intent(in) :: x
type(c_ptr), intent(in) :: data
type(f_data), pointer :: d
call c_f_pointer(data, d)
y = d%a * sin(d%k * x)
end function

subroutine foo(a, k)
real(dp) :: a, k
type(f_data), target :: data
data%a = a
data%k = k
print *, simpson(f, 0._dp, pi, c_loc(data))
print *, simpson(f, 0._dp, 2*pi, c_loc(data))
end subroutine

end module
```

As always, with the advantages of such re-casting, as Fortran lets you do if you really want to, come also the disadvantages that fewer compile- and run-time checks are possible to catch errors; and with that, inevitably more leaky, bug-prone code. So one always has to balance the costs and benefits.

Usually, in the context of scientific programming, where the main thrust is to represent and solve precise mathematical formulations (as opposed to create a GUI with some untold number of buttons, drop-downs, and other interface elements), simplest, least bug-prone, and fastest is to use one of the previous approaches.

VI: transfer() Intrinsic Function

Before Fortran 2003, the only way to do type casting was using the `transfer` intrinsic function. It is functionally equivalent to the method V, but more verbose and more error prone. It is now obsolete and one should use the method V instead.

Examples:

http://jblevins.org/log/transfer

http://jblevins.org/research/generic-list.pdf

http://www.macresearch.org/advanced_fortran_90_callbacks_with_the_transfer_function

VII: Object Oriented Approach

The module:

```
module integrals

use types, only: dp
```

```
implicit none
private

public :: integrand, simpson

! User extends this type
type, abstract :: integrand
contains
    procedure(func), deferred :: eval
end type

abstract interface
    function func(this, x) result(fx)
    import :: integrand, dp
    class(integrand) :: this
    real(dp), intent(in) :: x
    real(dp) :: fx
    end function
end interface

contains

real(dp) function simpson(f, a, b) result(s)
class(integrand) :: f
real(dp), intent(in) :: a, b
s = ((b-a)/6) * (f%eval(a) + 4*f%eval((a+b)/2) + f%eval(b))
end function

end module
```

The abstract type prescribes exactly what the integration routine needs, namely a method to evaluate the function, but imposes nothing else on the user. The user extends this type, providing a concrete implementation of the eval type bound procedure and adding necessary context data as components of the extended type.

Usage:

```
module example_usage

use types, only: dp
use integrals, only: integrand, simpson
implicit none
private

public :: foo

type, extends(integrand) :: my_integrand
    real(dp) :: a, k
contains
    procedure :: eval => f
end type

contains

function f(this, x) result(fx)
class(my_integrand) :: this
real(dp), intent(in) :: x
real(dp) :: fx
```

```
fx = this%a*sin(this%k*x)
end function

subroutine foo(a, k)
real(dp) :: a, k
type(my_integrand) :: my_f
my_f%a = a
my_f%k = k
print *, simpson(my_f, 0.0_dp, 1.0_dp)
print *, simpson(my_f, 0.0_dp, 2.0_dp)
end subroutine

end module
```

Complete Example of void * vs type(c_ptr) and transfer()

Here are three equivalent codes: one in C using `void *` and two codes in Fortran using `type(c_ptr)` and `transfer()`:

Language	Method	Link
C	`void *`	https://gist.github.com/1665641
Fortran	`type(c_ptr)`	https://gist.github.com/1665626
Fortran	`transfer()`	https://gist.github.com/1665630

The C code uses the standard C approach for writing extensible libraries that accept callbacks and contexts. The two Fortran codes show how to do the same. The `type(c_ptr)` method is equivalent to the C version and that is the approach that should be used.

The `transfer()` method is here for completeness only (before Fortran 2003, it was the only way) and it is a little cumbersome, because the user needs to create auxiliary conversion functions for each of his types. As such, the `type(c_ptr)` method should be used instead.