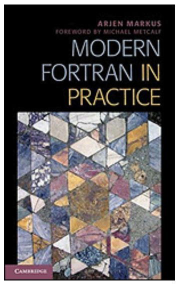

Chapter 5 - An Interface Problem
Modern Fortran in Practice
by Arjen Markus
Cambridge University Press © 2012 *Citation*

Recommend? yes no

Previous Next

5.3 Passing Extra Arguments

As an alternative to the previous approaches, have a look at what you can do with extra arguments. The integration routine does not use them directly because they are only needed by the function. The challenge is to pass them along.

Array of Parameters

The simplest solution is to pass an array of parameters:

```
subroutine integrate_trapezoid( f, params, xmin, xmax, &
                                steps, result )
    ...
    interface
        real function f(x, params)
            real, intent(in)                :: x
            real, dimension(:), intent(in)  :: params
        end function f
    end interface

    real, dimension(:) :: params
    ...
end subroutine
```

The function *f* can be implemented in a straightforward way:

```
real function f( x, params )
    real, intent(in)                :: x
    real, dimension(:), intent(in)  :: params

    f = exp(-params(1)*x) * cos(params(2)*x)

end function f
```

However, if the set of parameters does not only consist of reals, this solution is awkward [65]. In other situations than this numerical integration problem, you might be dealing with character strings or linked lists. Therefore, an alternative needs to allow more general data types.

Use the `transfer()` Function

In FORTRAN 77, there were very few facilities that could help, but with Fortran 90/95 you can use the `transfer()` function to convert arbitrary data into an array of reals and back:

```
type function_parameters
```

```
    real :: a
    real :: b
end type function_parameters

type(function_parameters) :: params

! Defines the type for the transfer function
real, dimension(1)        :: real_array
...
call integration_trapezoid( f, transfer(params,real_array), &
    xmin, xmax, steps, result )
...
```

While this works, it is not a very elegant solution: it puts the burden of converting the data on the user, even though you can hide it in an (internal) routine[4]:

```
program integrate

type function_parameters
    real :: a, b
end type function_parameters

type(function_parameters) :: params
...
call integration_trapezoid_ab( f, params, xmin, xmax, &
    steps, result )
...

contains
!
! This code can go into an include file if needed, to hide the
! details from sight
!
subroutine integration_trapezoid_ab( f, params, xmin, xmax, &
               steps, result )
    ...
    type(function_parameters) :: params

    ! Defines the type for the transfer function
    real, dimension(1)        :: real_array

    call integration_trapezoid(         &
        f, transfer(params,real_array), &
        xmin, xmax, steps, result )
end subroutine
end program integrate
```

Type-Bound Procedures

With Fortran 2003, you have more possibilities to solve this issue in an elegant way ([65], see also Chapter 11):

```
module integration_library

    implicit none

    type, abstract :: user_function
        ! No data - merely a placeholder
    contains
        procedure(function_evaluation), deferred, &
            pass(params) :: eval
    end type user_function

    abstract interface
        real function function_evaluation(x, params)
            import               :: user_function
            real                 :: x
            class(user_function) :: params
        end function function_evaluation
    end interface

contains

subroutine integrate_trapezoid( &
              params, xmin, xmax, steps, result )

    class(user_function)       :: params
    real, intent(in)           :: xmin, xmax
    integer, intent(in)        :: steps
    real, intent(out)          :: result

    integer                    :: i
    real                       :: x
    real                       :: deltx

    if ( steps <= 0 ) then
        result = 0.0
        return
    endif

    deltx = (xmax - xmin) / steps

    result = ( params%eval(xmin) + params%eval(xmax) )/ 2.0

    do i = 2,steps
        x      = xmin + (i - 1) * deltx
        result = result + params%eval(x)
    enddo

    result = result * deltx
end subroutine integrate_trapezoid
end module integration_library
```

The preceding module is shown in the following example. Note that the implementation of function *f* is now a part of the type `user_function`:

```
module functions
    use integration_library

    implicit none

    type, extends(user_function) :: my_function
        real :: a
        real :: b
    contains
        procedure, pass(params) :: eval => f
    end type my_function
```

```
contains
real function f( x, params )

    real, intent(in)    :: x
    class(my_function) :: params

    f = exp(-params%a*x) * cos(params%b*x)

end function f

end module functions
```

Rather than pass the name of the function, you now pass the *derived type* that contains the function you want to integrate:

```
program test_integrate

    use integration_library
    use functions

    implicit none

    type(my_function) :: params

    real               :: xmin, xmax, result
    integer            :: steps

    params%a = 1.0
    params%b = 2.0

    xmin     = 1.0
    xmax     = 10.0
    steps    = 10

    call integrate_trapezoid( params, xmin, xmax, steps, &
                              result )

    write(*,*) 'Result: ', result

end program test_integrate
```

The abstract derived type `user_function` provides the common type that the integration library uses for passing the function and its data. You need to define a specific implementation of that type in order to actually do the computation. The only thing that "feels" awkward about this solution is that each function to integrate requires its own type. You might say: the solution is data-centered instead of function-centered.

Procedure Pointers

As in the previous example, if instead of a type-bound procedure you use a procedure pointer, you can change the function that needs to be evaluated – without introducing a new type of each function. For this, you move the procedure component to the "data" section and add the pointer attribute:

```
module integration_library

    implicit none

    type, abstract :: function_parameters
        procedure(eval), pointer, pass(params) :: feval
    end type function_parameters

    abstract interface
        real function eval(x, params)
            import :: function_parameters
            class(function_parameters) :: params
        end function eval
    end interface
contains

subroutine integrate_trapezoid( &
              params, xmin, xmax, steps, result )

    interface
        real function f( x, params )
            import function_parameters
            real, intent(in)           :: x
            class(function_parameters) :: params
        end function f
    end interface

    class(function_parameters) :: params

    ... (identical to the previous implementation) ...

end subroutine integrate_trapezoid

end module integration_library
```

Now, you can vary the function that is to be integrated without introducing a new type for each function:

```
module functions
    use integration_library

    implicit none

    type, extends(function_parameters) :: my_parameters
        real :: a
    end type my_parameters

contains
real function f( x, params )

    real, intent(in)     :: x
    class(my_parameters) :: params

    f = exp(-params%a*x) * cos(params%b*x)

end function f

!
! Function g() does not use parameter b,
! but otherwise it has the same data requirements, hence
! reuse type "my_parameters".
!
real function g( x, params )

    real, intent(in)     :: x
    class(my_parameters) :: params
    g = params%a * x

end function g

end module functions

program test_integrate

    use integration_library
    use functions

    implicit none

    type(my_parameters) :: params
    real                :: xmin, xmax, result
    integer             :: steps

    params%a     = 1.0
    params%b     = 2.0

    xmin         = 1.0
    xmax         = 10.0
    steps        = 10

    params%feval => f   ! First function

    call integrate_trapezoid( &
             params, xmin, xmax, steps, result )
    write(*,*) 'Result f: ', result

    params%feval => g ! Second function

    call integrate_trapezoid( &
             params, xmin, xmax, steps, result )
    write(*,*) 'Result g: ', result

end program test_integrate
```

---

[4]You might call this and the solution with the internal routine the *Façade pattern* [54].

Previous Next