PART II – ADVANCED FORTRAN

Recommended Books

Chapman, S.J., 2017, *Fortran For Scientists and Engineers* (4th Ed.), McGraw-Hill.
Metcalf, M., Reid, J. and Cohen, M., 2018, *Modern Fortran Explained*, OUP.

12. User-Defined Types

See Sample Programs E

The intrinsic data types are `integer`, `real`, `complex`, `character`, `logical`. However, you can devise your own *user-defined* or *composite* or *derived* data types. Together with type-bound procedures ("methods"), type-extension ("inheritance") and public/private access, these yield the Fortran version of *classes* in object-oriented programming: see Section 18.

For example you might define a `type(student)` and a `type(vector)`:

```
type student
   character(len=20) surname
   character(len=20) firstname
   integer year
   real mark
end type

type vector
  real x, y, z
end type vector
```

Once declared, user-defined types behave in many ways like the built-in types. In particular:

- Variables `a`, `b`, `c` can be declared of this type by
```
type(student) a, b, c
```

- You can have arrays of user-defined types; e.g.
```
type(student), dimension(200) :: MACE
type(student), allocatable :: EE(:)
```

- They can be passed as procedure arguments, or returned from functions:
```
type(vector) function add( u, v )
   type(vector), intent(in) :: u, v
   plus%x = u%x + v%x
   plus%y = u%y + v%y
   plus%z = u%z + v%z
end function add
```

  (Later we shall see that this can be associated with the + operator for this particular type, making coding even easier.)

As illustrated in the last example, a user-defined type has multiple components, which are obtained by use of the % *component selector* [1]; thus, `u%x`, `u%y` and `u%z` for the `x`, `y` and `z` components of the `vector` type `u`.

Variables of derived type can be assigned either by specifying individual components; e.g. for our `student` type:

```
a%surname = "Apsley"
a%firstname = "David"
a%year = 2
a%mark = 75.0
```

or, considerably more efficiently, as a single *type-constructor* statement:

```
a = student( "Apsley", "David", 2, 75.0 )
```

Derived types may also be default-initialised – in whole or in part – in the type definition:

```
type(vector):
   real x
   real y
   real :: z = 0
end type vector
```

A composite variable of derived type may be used as a single entity or via its individual components; e.g.

```
if ( a%mark > 69.5 ) print *, "First class"
```

The components of a derived type can be any predefined type, including intrinsic types, arrays (including allocatable arrays), other derived types, or pointers (see later).

[1] C and C++ use a dot as the component selector, rather than %. Thus: `a.surname`, `a.firstname` etc.

It is common to declare types (and associated procedures and operators) in a module, so that they can be accessed by many program units via a `use` statement. The module may also `contain` procedures naturally associated with those types, as in the following example. Here there are new `point` and `polygon` types, with the latter incorporating an allocatable array of the former to allow for an arbitrary number of vertices. The module also contains utility routines:

`distance`	distance between two points
`perimeter`	perimeter of a polygon
`area`	area of a polygon

The following formula is used for area of a polygon defined by the coordinates of its vertices (taken in order):

$$A = \frac{1}{2}\sum_{i=1}^{n}(x_{i-1}y_i - x_i y_{i-1})$$

Here, (x_0, y_0) is tacitly equated with (x_n, y_n) to close the polygon. (According to this formula, the area is positive if the polygon is traversed anticlockwise, negative if traversed clockwise; the area function here just returns the absolute value.)

```
module Geometry
   implicit none

   type point
      real x, y
   end type point

   type polygon
      integer n
      type(point), allocatable :: p(:)
   end type polygon

contains

   real function distance( p, q )
      type(point), intent(in) :: p, q

      distance = sqrt( ( q%x - p%x ) ** 2 + ( q%y - p%y ) ** 2 )

   end function distance

   !-----------------------------------

   real function perimeter( poly )
      type(polygon), intent(in) :: poly
      integer i, n

      n = poly%n
      perimeter = distance( poly%p(n), poly%p(1) )
      do i = 2, n
         perimeter = perimeter + distance( poly%p(i-1), poly%p(i) )
      end do

   end function perimeter

   !-----------------------------------

   real function area( poly )
      type(polygon), intent(in) :: poly
      integer i, n

      n = poly%n
      area = poly%p(n)%x * poly%p(1)%y - poly%p(1)%x * poly%p(n)%y
      do i = 2, n
         area = area + poly%p(i-1)%x * poly%p(i)%y - poly%p(i)%x * poly%p(i-1)%y
      end do
      area = abs( area / 2 )
```

```
   end function area

end module Geometry

!============================================================

program test
   use Geometry
   implicit none
   type(polygon) triangle

   ! Note the automatic allocation of the p array component of a polygon on assignment
   triangle = polygon( 3,[ point( 0.0, 0.0 ), point( 3.0, 0.0 ), point( 0.0, 4.0 ) ] )

   write( *, * ) "Perimeter = ", perimeter( triangle )
   write( *, * ) "Area = ", area( triangle )

end program test
```

13. Interfaces

See Sample Programs E

13.1 Explicit Interfaces

For code to call some procedure (function or subroutine) it is often necessary for the compiler to know the *interface* to that procedure; i.e. the names and types of dummy arguments and, if a function, the return type. (This is similar to the *function prototypes* required in C and C++.) Examples where this is the case include:

- keyword or optional arguments;
- passing array or character arguments with variable size;
- passing pointer arguments;
- passing arguments which are procedures (or pointers to procedures);
- returning an array;
- when using generic functions;
- when using external routines in other languages (notably C and C++).

Such an *explicit interface* is usually provided by one of the following:

(i) Automatically, if the procedure is an internal subprogram; i.e. contained in the same program unit. Since the program unit is compiled as a whole, nothing further is required.

(ii) By providing a specific `interface block` in the calling routine.

(iii) By putting the procedure in a module, with a `use` statement for that module in the calling routine. **This is usually the most convenient way.**

Even when an explicit interface is not strictly required (e.g. when passing procedures as arguments, which can be done with an implicit interface – see Section 17) it does no harm, and it will reduce the chance of errors because it allows the compiler to check argument lists when invoking routines, rather than relying on the user to just "get it right".

13.2 Interface Blocks

These are placed in the specification (i.e. declarations) area of a program unit. If they are to be used in many places they can conveniently be placed in modules, so that they need only be declared once, and access may be achieved by a `use` statement.

They take the form

```
interface
```

procedure prototypes

```
end interface
```

For example, an interface block incorporating the function and subroutine to find radius in Section 9 might look like the following. Note that it tells the compiler how the function or subroutine is invoked (the number and types of arguments, and any function return type) but doesn't include the executable statements.

```
interface
   real function radius( a, b )
      real a, b
   end function radius

   subroutine distance( a, b, r )
      real a, b, r
   end subroutine distance
end interface
```

In this instance the interface is overkill. It would be simpler just to put the procedures in a module and `use` that module, thus providing an explicit interface.

Sometimes the procedures in the interface need access to variables from the outer module (for example, to declare kind parameters, or derived types). Even if inside a module they do not automatically have access to these by host association, so an import statement is necessary to bring those into scope. (See Section 18 for particular examples.)

```
      import point
```

13.3 Abstract Interfaces

In the above, we defined an interface for external procedures that actually exist. In many applications (particularly Sections 16-18) we wish to define prototypes for the form, but not necessarily the name, of procedures. For example, we might want a generic "`real` function with two `real, intent(in)` arguments", or "`logical` function with one `real, intent(in)` argument" or "subroutine with one `integer, intent(in)` and one `real, intent(out)` argument". For such an application we require an *abstract interface*. e.g.

```
abstract interface
   real function binaryOperator( a, b )
      real, intent(in) :: a, b
   end function binaryOperator

   logical function predicate( a )
      integer, intent(in) :: a
   end function predicate

   subroutine qualify( i, r )
      integer, intent(in) :: i
      real, intent(out) :: r
   end subroutine qualify
end interface
```

Then we can declare the form of *procedures* in much the same way as we declare the type of derived variables:

```
procedure(binaryOperator) add, multiply
procedure(predicate) isValid
```

This is particularly important for procedure pointers (Section 16 and 17).

13.4 Procedure Overloading – Generic Names

Generic (or *overloaded*) *procedures* occur where the same generic name can be used for several different argument types.[2] For example, the intrinsic function `abs( x )` can be called by the generic name `abs` regardless of whether the argument type is `integer`, `real`, `complex`, …. In fact, no such individual function actually exists: the compiler simply calls whichever procedure `iabs( x )`, `dabs( x )`, `cabs( x )` is appropriate for the type of argument `x`.

The general form is

`interface` *generic_name*

procedure prototypes *and/or* `[module] procedure` *statements*

`end interface` *generic_name*

For example, one might have:

```
interface squared
   integer function Isquared( i )
      integer i
   end function Isquared

   real function Rsquared( r )
      real r
   end function Rsquared
end interface squared
```

Here, the functions `Isquared` and `Rsquared` are defined outside the program unit containing the generic interface, whilst the latter contains just their prototypes. The more common situation, however, is that this generic interface is inside a module which also contains these functions. An explicit interface for these already exists in the module and is not permitted to be repeated. In this circumstance, we simply use a `module procedure` statement instead of the full prototypes; thus:

[2] C++ has *template* functions to do something akin to this. It is high on my (and many others') wish-lists for Fortran!

```
   interface squared
      module procedure Isquared
      module procedure Rsquared
   end interface squared
```

The word `module` in `module procedure` is optional but useful if the procedures are actually in the module. It should not be used for external procedures (i.e. not in the module). For those we can write

`procedure` *name*

provided that they already have an explicit interface in this program unit (e.g. from a previous interface statement); otherwise we would provide a full prototype at this point.

Whilst the type and number of arguments may differ, all procedures for a particular generic name must be of the same form: all functions or all subroutines.

A full program illustrating this is given below.

```
module GenericRoutines
   implicit none

   interface squared                   ! Generic interface
      module procedure Isquared
      module procedure Rsquared
   end interface squared

contains
   integer function Isquared( x )      ! Function for an integer
      integer x
      Isquared = x * x
   end function Isquared

   real function Rsquared( x )         ! Function for a real
      real x
      Rsquared = x * x
   end function Rsquared

end module GenericRoutines


program main
   use GenericRoutines
   implicit none
   integer :: i = 3
   real :: r = 2.5

   ! The appropriate function is called via the single interface squared()
   print *, "The square of integer ", i, " is ", squared( i )
   print *, "The square of real "   , r, " is ", squared( r )

end program main
```

13.5 Operators

One can extend the built-in operators `+`, `–`, `*`, `/` etc. to our own derived types. We can also create new operations for intrinsic types (`integer`, `real`, `complex`, `character`, `logical`). This is done by associating an operator symbol with the procedures defining it via an interface statement of the form

```
interface operator ( op )

    procedure prototypes
        and/or
    [module] procedure statements

end interface
```

where *op* is one of the predefined operators and the procedures must be functions with the requisite number of arguments: 2 for binary operations, 1 for unary operations. These arguments should have `intent(in)`. Predefined operator symbols for derived types have the same precedence as they would have for intrinsic type (`*` before `+`, for example).

We can also define our own named operators as *op* if we prefix and postfix a dot: e.g. `.dot.` or `.cross.`

By far the most common place to put these interfaces and related procedures is in modules, so that the operations may be made available by a simple `use` statement.

In the following example note the types of the arguments and the return value of the function. For example, the `.dot.` operator is defined by a procedure `dotProd` taking two arguments of user-defined `type(vector)` and returning a `real` value, whilst the `.cross.` operator is defined by a procedure `crossProd` taking two arguments of user-defined `type(vector)` and returning a `type(vector)`. Note that the operator symbol is just a (convenient) alias; for example,

```
a .dot. b
```

will give exactly the same result as

```
dotProd( a, b )
```

```
module exampleOperators
   implicit none

   ! User-defined type
   type vector                          ! Defines a vector with 3 components
      real x, y, z
   end type vector

   ! Some new operators and the functions for which they are shorthand
   interface operator( + )              ! Add two vectors
      module procedure plus
   end interface

   interface operator( * )              ! Multiply a vector by a scalar
      module procedure scalarTimes
   end interface

   interface operator( .dot. )          ! Dot product of two vectors
      module procedure dotProd
   end interface

   interface operator( .cross. )        ! Cross product of two vectors
      module procedure crossProd
   end interface

contains

   type(vector) function plus( u, v )
      type(vector), intent(in) :: u, v
      plus%x = u%x + v%x
      plus%y = u%y + v%y
      plus%z = u%z + v%z
   end function plus

   type(vector) function scalarTimes( r, u )
      real, intent(in) :: r
      type(vector), intent(in) :: u
      scalarTimes%x = r * u%x
      scalarTimes%y = r * u%y
      scalarTimes%z = r * u%z
   end function scalarTimes

   real function dotProd( u, v )
      type(vector), intent(in) :: u, v
      dotProd = u%x * v%x + u%y * v%y + u%z * v%z
   end function dotProd

   type(vector) function crossProd( u, v )
      type(vector), intent(in) :: u, v
      crossProd%x = u%y * v%z - u%z * v%y
      crossProd%y = u%z * v%x - u%x * v%z
      crossProd%z = u%x * v%y - u%y * v%x
   end function crossProd

end module exampleOperators

!==============================================================

program main
   use exampleOperators
   implicit none
   type(vector) :: a = vector( 2.0, 4.0, 1.0 ), b = vector(-1.0, 3.0, 7.0 )

   write( *, * ) "a     = ", a
   write( *, * ) "b     = ", b
   write( *, * ) "a + b = ", a + b
   write( *, * ) "5a    = ", 5.0 * a
   write( *, * ) "a.b   = ", a .dot. b
   write( *, * ) "axb   = ", a .cross. b

end program main
```

```
 a     =    2.0000000   4.0000000   1.0000000
 b     =   -1.0000000   3.0000000   7.0000000
 a + b =    1.0000000   7.0000000   8.0000000
 5a    =   10.0000000  20.0000000   5.0000000
 a.b   =   17.0000000
 axb   =   25.0000000 -15.0000000  10.0000000
```

14 Advanced Procedures

See Sample Programs F

14.1 Keyword and Optional Arguments

For many procedures, the argument list can be very long and, in many circumstances, a lot of arguments may not actually be used, or default values would be perfectly adequate. In this case it is convenient to be able to:

- associate *actual arguments* (values supplied by the calling function) with *dummy arguments* (those appearing in the procedure code) by *name* rather than their position in the argument list; these are called *keyword arguments*;
- optionally omit some of the arguments, allowing the procedure, if necessary, to give default values to those variables itself; these are called *optional arguments*.

For example, consider an incredibly verbose statement to open a file:

```
open( unit=10, file="input.txt", access="sequential",     &
      action="read", form="formatted", asynchronous="no", &
      status="old", iostat=io, encoding="default" )
```

(Believe it or not, this is only a fraction of the keywords that could actually be used!) In many circumstances, we could achieve exactly the same objective with the simple statement and with one positional and one keyword argument:

```
open( 10, file="input.txt" )
```

The arguments associated with the other keywords are either not used or their defaults would be perfectly adequate.

Note:

- Keyword and optional arguments can only be used if an explicit interface for that procedure is available at the point of call. The most convenient method is to put the procedure in a module and access it by a `use` statement.
- If used, the keywords are the names of the dummy arguments in the procedure's code (or, strictly, in the explicit interface).
- Arguments passed with keywords (*keyword* = *argument_value*) can be in any order.
- No more positional arguments can be passed after the first keyword (as the compiler wouldn't be able to work out which they were).
- Dummy arguments identified purely by position can be given the `optional` attribute and the corresponding actual arguments may be omitted. (Without this they *must* be there).
- The procedure can call the logical function
  `present(` *argument_name* `)`
  to determine if that argument was actually supplied.

Keyword arguments can also be used in type constructors.

An example (overleaf) shows the mixing of primary colours in light. Each of red, green, blue is 0 or 1: if they are absent they default to 0.

```
module RGB
   implicit none

contains
   character(len=7) function colour( red, green, blue )
      implicit none
      integer, optional, intent(in) :: red, green, blue
      character(len=7), parameter :: colourMap(0:7) = &
                                  [ "black  ", "red    ", "green  ", "yellow ", &
                                    "blue   ", "magenta", "cyan   ", "white  " ]
      integer r, g, b

      r = 0;   if ( present( red   ) .and. red   /= 0 ) r = 1
      g = 0;   if ( present( green ) .and. green /= 0 ) g = 1
      b = 0;   if ( present( blue  ) .and. blue  /= 0 ) b = 1
      colour = colourMap( 1 * r + 2 * g + 4 * b )

   end function colour

end module RGB

!============================================================

program main
   use RGB                                          ! Provides explicit interface
   implicit none

   write( *, * ) colour()                           ! none
   write( *, * ) colour( blue = 1, green = 1 )      ! blue + green
   write( *, * ) colour(        1, green = 1 )      ! red + green (positional red)
   write( *, * ) colour( red  = 0, blue  = 1 )      ! blue
   write( *, * ) colour( 1, 1 )                     ! red + green (positional)
   write( *, * ) colour( 1, 1, 1 )                  ! red + green + blue

end program main
```

```
black
cyan
yellow
blue
yellow
white
```

14.2 Recursive Procedures

Recursion means a procedure calling itself. The classic example is a factorial:

$n! = n \times (n\text{-}1)!$ with $1! = 1$

Note that here we have a *recursion*:

factorial(n) = n * factorial(n-1)

but, to prevent the recursion going on indefinitely, we also need terminating *base case*(*s*) where no further call to the function appears on the RHS:

factorial(1) = 1.

Recursion is easily implemented in Fortran simply by:

- Prefixing the `function` or `subroutine` statement by the `recursive` keyword.
- For the `function` form, returning the output via a `result` clause rather than function name.

Subroutine and function forms, with accompanying main driver programs are given below. Note that factorials grow very fast and will rapidly overflow a default integer, so don't enter anything too large!

Function form:

```
program test
   implicit none
   integer, external :: factorial
   integer n

   print *, "Enter n: "
   read *, n
   print *, "Factorial is ", factorial( n )

end program test

!=========================================

recursive integer function factorial( n ) result( answer )
  implicit none                        ! Note the result clause
  integer, intent(in) :: n

  if ( n <= 1 ) then                   ! Base cases:  0! = 1! = 1
     answer = 1
  else
     answer = n * factorial( n - 1 )   ! Recursion; note output in "answer"
  end if

end function factorial
```

Subroutine form:

```
program test
   implicit none
   external factorialSub
   integer n
   integer F

   print *, "Enter n: "
   read *, n
   call factorialSub( n, F )
   print *, "Factorial is ", F

end program test

!=========================================

recursive subroutine factorialSub( n, F )
  implicit none
  integer, intent( in  ) :: n
  integer, intent( out ) :: F

  F = 1
  if ( n <= 1 ) return                 ! Base case

  call factorialSub( n - 1, F )        ! Recursion
  F = n * F                            !

end subroutine factorialSub
```

Note that the type of the variable in the `result` clause (here named `answer`) must be given either

- implicitly in the function statement itself:
  `recursive` **`integer`** `function factorial( n ) result( answer )`
- or by a separate type statement:
  `recursive function factorial( n ) result( answer )`
  `...`
  `integer` `answer`

but not both. In the latter case, it is the `result` variable that has its type declared, not the name of the function.

The `result` clause can, in fact, be used in any function.

Other recursive routines are given in the examples. Advanced examples include the *quicksort* algorithm, *back-tracking*, and data storage structures such as *binary trees*.

14.3 Elemental Procedures

One of Fortran's most powerful features is its natural handling of arrays. Most intrinsic function are *elemental*; that is, they can be applied equally to scalars (single variables) or arrays (indexed collections of variables). Consider, for example,

```
sin(A)
```

If `A` is a `real` scalar then it will return a single real number – the sine of that angle in radians. However, if `A` is an array, then it will return an array of the same shape whose elements are the sines of the corresponding elements of `A`.

User procedures can be made to behave in the same way by prefixing them by the keyword `elemental`. They must have an explicit interface at the point of call. The easiest way to achieve this is to put them in a module, with a `use` statement for that module in the calling program.

```
module ElementalModule
   implicit none

contains

   elemental real function squared( x )     ! Procedure declared "elemental"
      real, intent(in) :: x

      squared = x * x

   end function squared

end module ElementalModule

!==================================================

program test
   use ElementalModule                    ! Provides explicit interface
   implicit none
   real :: x = 10.0                       ! Scalar
   integer i
   real :: A(5) = [ ( i, i = 1, 5 ) ]     ! Array

   write( *, * ) "Square of scalar is ", squared( x )
   write( *, * ) "Square of array A is ", squared( A )

end program test
```

```
Square of scalar is    100.000000
Square of array A is    1.00000000       4.00000000       9.00000000       16.0000000       25.0000000
```

15. Advanced Characters and Arrays

See Sample Programs F

Arrays and characters share the properties that:

- they have a *size* (for arrays, one or more *dimensions*; for characters, a *length*)
- they can be *allocatable* (i.e. they may be given a size at run time).

Character arrays are also possible, with both length and dimensions.

These present particular issues with regard to:

- declaration;
- allocation;
- passing as procedure arguments.

15.1 Characters

15.1.1 Explicit Length Declaration (`len=`*n*)

The following are equivalent (but the first is my preference):

```
character(len=n) name
character(n) name
character*n name
```

(In addition to `len`, characters also have a second type parameter – `kind` – for systems where additional character sets are available. This isn't very portable and won't be considered here.)

Example:

```
character(len=20) university
university = "Cambridge"     ! Note: pads with 11 blanks
```

15.1.2 Assumed Length Declaration (`len=*`)

The length can be declared as an asterisk * in two cases where the actual length can be assumed from the context.

(i) Declaration of a named constant (i.e. with the `parameter` attribute); e.g.

```
character(len=*), parameter :: country = "England"
character(len=*), parameter :: dataFormat = "( i3, 1x, f10.3 )"
```

(ii) Dummy argument of a procedure; e.g.

```
subroutine grossDomesticProduct( country )
   character(len=*), intent(in) :: country
```

The length will be inferred from the actual arguments when the procedure is invoked.

15.1.3 Deferred Length Declaration (`len=:`)

This indicates a character whose length can be allocated, either in that subprogram or in any procedure to which it is passed as an argument.

One method of allocation is, e.g.

```
character(len=:), allocatable :: university
. . .
allocate( character(len=20) :: university )
university = "Oxford"       ! Character length 20, so pads with 14 blanks
. . .
deallocate( university )
```

Note that, as of Fortran 2003, allocation (and reallocation) can be done automatically by assignment (but not in the type declaration statement itself):

```
character(len=:), allocatable :: university
university = "Oxford"       ! Automatically allocates to length 6
university = "Cambridge"    ! Automatically reallocates to length 9
```

Moreover, local variables are (unless given the `save` attribute) automatically deallocated at the end of the subprogram in which they were allocated; there is no need for an explicit `deallocate` statement.

If passed as a procedure argument then the interface to that procedure must be explicit at the point of call (e.g. by putting the procedure in a module, with a `use` statement for that module in the calling routine). The length of the character can be determined at any time by use of the `len()` intrinsic function.

15.1.4 Automatic Character Variables

Automatic variables are local variables of character or array type in procedures whose size is determined at run time by procedure arguments, either directly (i.e. the size is passed as a procedure argument) or indirectly (i.e. the size is inferred by a call to `len()` for characters or `size()` for arrays).

Memory is set aside for the variable on entry to the procedure and recovered on return.

As the size is determined from a procedure argument, automatic variables can only be used in subroutines and functions, not in main programs.

In many circumstances it may be a toss-up whether to use an automatic variable or allocatable ones (but only the latter is possible in a main program).

Example:

```
subroutine reverse( word )
   implicit none
   character(len=*), intent(inout) :: word  ! Assumed-length character variable
   character(len=len(word) ) copy           ! Automatic variable, length from argument
   integer i, j, L

   L = len( word )
   do i = 1, L
      j = L - i + 1
      copy(i:i) = word(j:j)                 ! Note: single character
   end do

   word = adjustl( copy )                   ! Left-align in character variable

end subroutine reverse

!============================================================

program main
   implicit none
   character(len=20) text
   character(len=*), parameter :: fmt = "( a )"

   write( *, fmt, advance="no" ) "Enter some text: "
   read( *, fmt ) text

   call reverse( text )

   write( *, fmt ) "Reversed text:   " // text

end program main
```

15.2 Arrays

15.2.1 Explicit-Shape Arrays (`dimension(`*n*`)`)

For these the dimensions are either known constants or, for procedure arguments, passed explicitly as subroutine arguments.

Example declarations:
```
integer A(20)           ! rank 1 (one-dimensional) array
real B(0:5,0:10)        ! rank 2 (two-dimensional) array
```
or, equivalently,
```
real, dimension(0:5,0:10) :: B
```

Example of procedure arguments:
```
subroutine example( C, n )
   integer, intent(in) :: n
   real, intent(inout) :: C(n)      ! Explicit shape, rank 1 of dimension n
```

15.2.2 Assumed-Shape Arrays (`dimension(:)`)

For these the rank is known, but dimensions are not known at compile time, so are *deferred* until later. This may occur for:

- allocatable arrays;
- arrays (allocatable or not) passed as procedure arguments or returned as function values.

In the latter case, array sizes can be determined by using `size()` or other intrinsic function – see later – in the procedure to which it is passed, provided that there is an explicit interface at the point of call; (usually by putting the procedure in a module, with a `use` statement for that module in the calling routine.) If the array is to be allocated in the procedure then both dummy and actual arguments must have the `allocatable` attribute.

Note that it is the *extent* in each dimension which is passed, not the upper and lower bounds. The default lower bound in the receiving program is 1, and if it is to be anything else – e.g. 0 – then that must be stated explicitly; e.g. by
```
0:
```

The following example shows Gaussian elimination with partial pivoting to solve a system of linear equations

Ax = **b**

Note the assumed-shape arrays in the arguments `A`, `B` and `X`, and the allocatable arrays `AA` and `BB` which are to be used as local copies to avoid changing the original matrix and vector:
```
logical function GaussElimination( A, B, X )
   real(kind=dp), intent(in ) :: A(:,:), B(:)          ! Assumed-shape arrays
   real(kind=dp), intent(out) :: X(:)                  ! Assumed-shape array
   real(kind=dp), allocatable :: AA(:,:), BB(:)        ! Local allocatable arrays
```

Note also the extensive use of array sections; e.g.
```
AA(r,i:n) = AA(r,i:n) - multiple * AA(i,i:n)
```
and elemental routines:
```
call swap( AA(i,i:n), AA(r,i:n) )
```

```
module matrix
   implicit none
   integer, parameter :: dp = kind( 1.0d0 )                   ! Double precision
   real(kind=dp), parameter :: SMALL    = 1.0e-10             ! Used to check matrix singularity
   real(kind=dp), parameter :: NEARZERO = 1.0e-10             ! Used to filter output

contains

   !-----------------------------------------------------

   logical function GaussElimination( A, B, X )               ! Solve AX = B by Gaussian elimination
      real(kind=dp), intent(in ) :: A(:,:), B(:)              ! Assumed-shape arrays
      real(kind=dp), intent(out) :: X(:)                      ! Assumed-shape array
      real(kind=dp), allocatable :: AA(:,:), BB(:)            ! Local allocatable arrays
      integer n
      integer i, k, r
      real (kind=dp) val, maxA, pivot, multiple

      GaussElimination = .false.
      n = size( A, 1 )
      AA = A;   BB = B                                        ! Automatic allocation on assignment

      ! Row operations for i = 1 ,,,, n   (n only needed to check for singularity)
      do i = 1, n - 1
         ! Pivot: find row r below with largest element in column i
         r = i
         maxA = abs( AA(i,i) )
         do k = i + 1, n
            val = abs( AA(k,i) )
            if ( val > maxA ) then
               r = k
               maxA = val
            end if
         end do
         if ( r /= i ) then
            call swap( AA(i,i:n), AA(r,i:n) )                 ! Note array sections and elemental swap
            call swap( BB(i), BB(r) )
         end if

         ! Row operations to make upper-triangular
         pivot = AA(i,i)
         if ( abs( pivot ) < SMALL ) return                   ! Singular matrix
         do r = i + 1, n                                      ! On lower rows
            multiple = AA(r,i) / pivot                        ! Multiple row i to clear element in ith column
            AA(r,i:n) = AA(r,i:n) - multiple * AA(i,i:n)      ! Note array sections
            BB(r) = BB(r) - multiple * BB(i)
         end do
      end do

      ! Back-substitute
      do i = n, 1, -1
         X(i) = ( BB(i) - sum( AA(i,i+1:n) * X(i+1:n) ) ) / AA(i,i)
      end do

      GaussElimination = .true.                               ! If we get here we are successful
                                                              ! AA and BB are automatically deallocated here
   end function GaussElimination

   !-----------------------------------------------------

   subroutine vectorWrite( title, V )                         ! Write a vector
      character(len=*), intent(in) :: title
      real(kind=dp), intent(in) :: V(:)                       ! Assumed-shape array
      character (len=*), parameter :: fmt = "( *( 1x, g11.4 ) )"
      integer n
      integer i

      n = size( V, 1 )
      write( *, * ) title
      write( *, fmt ) ( filter( V(i) ), i = 1, n )
      write( *, * )

   end subroutine vectorWrite

   !-----------------------------------------------------

   subroutine matrixWrite( title, A )                         ! Write a matrix
      character(len=*), intent(in) :: title
      real(kind=dp), intent(in) :: A(:,:)                     ! Assumed-shape array
      character (len=*), parameter :: fmt = "( *( 1x, g11.4 ) )"
      integer m, n
```

```
      integer i, j

      write( *, * ) title
      m = size( A, 1 )
      n = size( A, 2 )
      do i = 1, m
         write( *, fmt ) ( filter( A(i,j) ), j = 1, n )
      end do
      write( *, * )

   end subroutine matrixWrite

   !--------------------------------------------------------

   elemental function filter( x )                          ! Filter near-zero. Note: elemental
      real(kind=dp), intent(in) :: x
      real(kind=dp) filter

      filter = x
      if ( abs( x ) < NEARZERO ) filter = 0.0

   end function filter

   !--------------------------------------------------------

   elemental subroutine swap( x, y )                       ! Swap items. Note: elemental
      real(kind=dp), intent(inout) :: x, y
      real(kind=dp) t

      t = x
      x = y
      y = t

   end subroutine swap

   !--------------------------------------------------------

end module matrix

!=======================================================================

program main
   use matrix
   implicit none
   integer, parameter :: N = 4
   real(kind=dp) :: A(N,N), B(N), X(N)

   A(1,:) = [  1.0,  2.0,  3.0,  4.0 ]
   A(2,:) = [ -2.0,  5.0,  5.0,  7.0 ]
   A(3,:) = [  1.0,  9.0, 10.0,  3.0 ]
   A(4,:) = [  2.0,  2.0,  4.0,  3.0 ]
   B      = [  1.0,  2.0,  3.0,  4.0 ]

   if ( GaussElimination( A, B, X ) ) then
      call matrixWrite( "A:", A )
      call vectorWrite( "B:", B )
      call vectorWrite( "X:", X )
      call vectorWrite( "Check AX (should equal B)", matmul( A, X ) )
   else
      write( *, * ) "Unable to solve"
   end if

end program main
```

```
A:
   1.000       2.000       3.000       4.000
  -2.000       5.000       5.000       7.000
   1.000       9.000       10.00       3.000
   2.000       2.000       4.000       3.000

 B:
   1.000       2.000       3.000       4.000

 X:
  -2.162      -4.459       4.676     -0.4865

 Check AX (should equal B)
   1.000       2.000       3.000       4.000
```

15.2.3 Assumed-Size Arrays (`dimension(*)`)

These may be used for arrays as procedure arguments. What is actually passed behind the scenes is a pointer to the start of the array, and the user can address the array elements as `A(1)`, `A(2)`, … provided he takes full responsibility for knowing how big the array is. This makes it perfectly possible – but definitely not a good idea! – to read or write beyond the end of the array.

OK: every book on Fortran that I own says that assumed-size arrays are an accident waiting to happen, but there is plenty of code out there (in both Fortran and C) using them.

15.2.4 Automatic Arrays

Automatic arrays can only be used in subroutines and functions, not main programs, since the array size is determined, directly or indirectly, from a procedure argument whose value is known only at run-time.

They are local and temporary arrays (unless they have the `save` attribute). Heap memory is awarded to them on entry to the routine and recovered on return.

In the last example we can use automatic arrays instead of local allocatable arrays simply by changing the type declaration of local copies `AA` and `BB`:

```
   real (kind=dp) :: AA(size(A,1),size(A,2)), BB(size(B))
```

(In this instance we rather hope that all the extents determined by the `size` function are the same.)

15.2.5 Detecting the Shape of Arrays

A number of intrinsic routines exist to query the shape of arrays. Assume `A` is an array with 1 or more dimensions. Its

rank is the number of dimensions;
extents are the number of elements in each dimension;
shape is the collection of extents.

The following routines are available to determine these:

`lbound( A )`	returns a rank-1 array holding the lower bound in each dimension.
`lbound( A, i )`	returns an integer holding the lower bound in the `i`th dimension.
`shape( A )`	returns a rank-1 array giving the extents in each direction
`size( A )`	returns an integer holding the complete size of the array (product of its extents)
`size( A, i )`	returns an integer holding the extent in the `i`th dimension.
`ubound( A )`	returns a rank-1 array holding the upper bound in each dimension.
`ubound( A, i )`	returns an integer holding the upper bound in the `i`th dimension.

15.2.6 Functions That Return Arrays

Functions can return any type of variable … including arrays and characters. These arrays can even be allocatable, as for the function `getArray()` in the following example.

```
module ArrayRoutines
   implicit none

contains

   !-----------------------------------

   function getArray()
      integer, allocatable :: getArray(:)        ! Declared return type
      integer n
      character(len=*), parameter :: fmt = "( a, i0, a )"

      write( *, fmt, advance="no" ) "How many numbers do you want? "
      read( *, * ) n
      allocate( getArray(n) )
      write( *, fmt, advance="no" ) "Enter ", n, " numbers separated by spaces: "
      read( *, * ) getArray

   end function getArray

   !-----------------------------------

   subroutine printArray( A, fmt )
      integer, intent(in) :: A(:)
      character(len=*) fmt

      write( *, fmt, advance="no" ) A
      write( *, * )
   end subroutine printArray

   !-----------------------------------

end module ArrayRoutines

!=========================================================

program test
   use ArrayRoutines
   implicit none
   integer, allocatable :: X(:)
   character(len=*), parameter :: fmt = "( *( i0, :, ' --> ' ) )"

   X = getArray()
   call printArray( X, fmt )

end program test
```

```
How many numbers do you want? 5
Enter 5 numbers separated by spaces: 10 20 30 40 50
10 --> 20 --> 30 --> 40 --> 50
```

15.3 Character Arrays

In

```
   character(len= …)
```

the `len=` element is part of the character type, not an array dimension. If `len=1` then it can be omitted: the default character length is 1; i.e. a single byte.

We may also have *character arrays* requiring both length and dimension to be specified. e.g.

```
   character(len=9) :: days(7)
```

or

```
   character(len=9), dimension(7) :: days
```

followed by, e.g.,

```
   days = [ "Sunday   ", "Monday   ", "Tuesday  ", "Wednesday", &
            "Thursday ", "Friday   ", "Saturday " ]
```

(If using an array constructor, as here, all elements must have the same length; i.e. we must pad each with the requisite number of blanks to give them the same length: Fortran does not allow *ragged* arrays for intrinsic types – although there are sneaky ways of doing it with components of derived types.)

Both length and dimension may be allocatable; e.g.

```
   character(len=:), allocatable :: days(:)
```

They can then be allocated explicitly:

```
   allocate( character(len=9) :: days(7) )
```

Alternatively, like any other allocatable array, they can be allocated by assignment:

```
program main
   implicit none
   character(len=:), allocatable :: days(:)

   days = [ "Sunday   ", "Monday   ", "Tuesday  ", "Wednesday", "Thursday ", "Friday   ", "Saturday " ]
   write( *, "( a )" ) days

end program main
```

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

15.4 Resizing Arrays

Often the maximum size of an array cannot be anticipated in advance, and an array may need to expand (or contract) at run-time. In contrast to C++'s STL vectors, Fortran users must code the resizing themselves, a typical strategy being to increase array size by a fixed factor when the top is reached (or decrease size if the used region of the array has contracted sufficiently).

To resize an array we must copy the data into a temporary array before deallocating and reallocating our main array. The following subroutine (assumed to be part of a module) does this for an integer array being resized to a new size, `n`. Note that it uses `move_alloc` to transfer the allocation (i.e. defined region of memory) from the temporary array back to the main array, so avoiding a large and potentially-expensive second copy operation.

```
subroutine resize( A, n )
   implicit none
   integer, intent(inout), allocatable :: A(:)
   integer, intent(in) :: n
   integer, allocatable :: temp(:)
   integer copySize

   allocate( temp(n) )
   copySize = min( n, size( A ) )
   temp(1:copySize ) = A(1:copySize)
   call move_alloc( temp, A )

end subroutine resize
```

16. Pointers

See Sample Programs G

16.1 Pointers and Targets

A pointer is a special type of variable that holds not a data value, but the memory address of another variable. Importantly, it can switch dynamically between the variables of a particular type that it points to.

A pointer may point to either:

- a declared variable or array, which must have the `target` attribute, or
- another pointer, or
- a new, unnamed piece of memory which is dynamically allocated at runtime and can only be accessed via that pointer.

Once *associated* with that pointer, the value at the memory location can be accessed by either the name of the target variable or the name of the pointer; that is, effectively the pointer becomes just an *alias* for that variable.[3]

When declaring a *pointer*, one must give it the `pointer` attribute and specify the type of variable that it can point to; e.g.

```
   real, pointer :: p
```

Any variable that it could be associated with requires the `target` attribute (to prevent an aggressively-optimising compilers from eliding that variable out of existence):

```
   real, target :: a, b
```

The association of a pointer with a specific target is effected by the => operator. e.g.

```
   p => a
```

This, together with the fact that the pointer can switch its allegiance, is illustrated below.

```
program example
   implicit none
   real, target :: a = 2.5, b = 19.1
   real, pointer :: p

   p => a
   write( *, * ) a, b, p, p * p      ! Writes 2.5, 19.1, 2.5, 6.25
   p => b
   write( *, * ) a, b, p, p * p      ! Writes 2.5, 19.1, 19.1, 364.81

end program example
```

In the example above, a pointer was declared that could point to a scalar variable. Pointers can also point to arrays, but that must be declared in the pointer's type statement, giving the appropriate rank (number of dimensions); e.g.

```
   integer, pointer :: p(:)
   real, pointer :: q(:,:)
   type(vector), pointer :: r(:)
```

Similarly, pointers can point to characters or character arrays, optionally with a colon **:** for the `len` parameter:

```
   character(len=:), pointer :: p(:)
```

```
program example
   implicit none
   character(len=3), target :: lower(12) = [ "jan", "feb", "mar", "apr", "may", "jun", &
                                             "jul", "aug", "sep", "oct", "nov", "dec" ]
   character(len=3), target :: upper(12) = [ "JAN", "FEB", "MAR", "APR", "MAY", "JUN", &
                                             "JUL", "AUG", "SEP", "OCT", "NOV", "DEC" ]
   character(len=3), pointer :: pc(:)
   integer ans

   write( *, * ) "Enter 1 for lower, 2 for upper"
   read( *, * ) ans
   select case( ans )
      case( 1 );   pc => lower;
      case( 2 );   pc => upper;
   end select

   write( *, "( 12( a, 1x ) )" ) pc

end program example
```

[3] Fortran is unusual in not requiring an explicit *dereferencing* operator. If variable `x` is pointed to by pointer `p` then in Fortran we may use the simple alias `p`; in C++ we need to write `*p`, where `*` is the dereferencing operator.

Whether a pointer is currently associated may be determined by the `associated()` function:

```
if ( .not. associated( p ) ) print *, "Pointer is not associated"
```

To test if it is associated with a specific variable then use the alternative form, e.g.,

```
if (associated( p, A ) ) print *, "Pointer is associated with A"
```

When a pointer is first declared in a type statement its association status is undefined, so it is common to initialise it with the universal `null()` pointer; thus:

```
integer, pointer :: p => null()
```

or one can use separate statements:

```
integer, pointer :: p, q
nullify( p, q )
```

A pointer may be a component of a derived data type; for example:

```
type mytype
   integer i
   real r
   integer, pointer :: ptr
end type mytype
```

If we use a type constructor to initialise such a type then initial assignment is replaced by initial association for the pointer component. e.g. for the type above, if m is declared of `type(mytype)`, then

```
m = mytype( 4, 2.6, j )
```

has the same effect as

```
m%i = 4
m%r = 2.6
m%r => j
```

16.2 Dynamic Data Structures

Pointers, scalar or array, can be `allocated` to point to unnamed memory locations at run time, thus allowing arbitrarily large data structures to be available, even if their size is not known in advance. As these are otherwise unnamed they may only be accessed via the pointer.

```
integer, pointer :: p      ! Single item
allocate( p )
```

or

```
real, pointer :: p (:)     ! Array
allocate( p(1000000) )
```

Note, however, that if the pointer is switched to point elsewhere before it is `deallocate`'d then that piece of memory effectively becomes inaccessible. Then we have a *memory leak*. Given enough of these a program will eventually run out of RAM and crash.

Many extremely useful data structures can be constructed using allocation of pointers. Classic examples are *linked lists* (illustrated below) and *binary trees* (illustrated in the examples).

In a linked list the basic unit is a user-defined type called `node`:

```
type node
   integer value
   type(node), pointer :: next
end type node
```

which simply consists of a value and a further pointer to the next node (if there is one) or the null pointer (if not). By successive allocations to the `next` pointer of each node, arbitrarily long chains can be built up, the whole being delimited by head and tail pointers to the nodes at the two ends:

```
type(node), pointer :: head, tail
```

The advantages of this type of structure are that additional nodes can be added at will, whilst elements may be deleted or swapped simply by disassociating and/or reassociating pointers, which is much more efficient than having to move many data items around.

A much more advanced linked-list class is given in the samples for Section 18 (object-oriented Fortran).

```
module LinkedList
   implicit none

   type node
      integer value
      type(node), pointer :: next
   end type node

   type(node), pointer :: head => null()
   type(node), pointer :: tail => null()

contains

   subroutine push_back( value )
      integer, intent(in) :: value
      type(node), pointer :: p

      allocate( p )                             ! Allocate a new node
      p = node( value, null() )                 ! ... and initialise (by type constructor)
      if ( associated( tail ) ) tail%next => p   ! Adjust tail pointer
      tail => p
      if ( .not. associated( head ) ) head => p  ! Only if list was originally empty

   end subroutine push_back

   !----------------------------------------------

   subroutine push_front( value )
      integer, intent(in) :: value
      type(node), pointer :: p

      allocate( p )                             ! Allocate a new node
      p = node( value, head )                   ! ... and initialise
      head => p                                 ! Adjust head pointer

      if ( .not. associated( tail ) ) tail => p  ! Only if list was originally empty

   end subroutine push_front

   !----------------------------------------------

   subroutine erase( value )
      integer, intent(in) :: value
      type(node), pointer :: p, prev;

      if ( .not. associated(head) ) return       ! If the list is empty

      do while ( head%value == value )           ! Values at the front of the list
         if ( .not. associated( head%next ) ) then
            deallocate( head )
            head => null()
            tail => null()
            return
         else
            p => head%next
            deallocate( head )
            head => p
         end if
      end do

      p => head%next                            ! Loop through the rest of the list
      prev => head
      do while( associated( p ) )
         if ( p%value == value ) then           ! If value found ...
            prev%next => p%next                 ! ... bypass node ...
            deallocate( p )                     ! ... and then delete it
         else
            prev => p                           ! Otherwise, just move on
         end if
         p => prev%next
      end do
```

```
   end subroutine erase

   !---------------------------------------------

   subroutine writeList( fmt )
      type(node), pointer :: p
      character(len=*) fmt

      p => head
      do while( associated( p ) )
         write( *, fmt, advance = "no" ) p%value
         p => p%next
      end do
      write( *, * )

   end subroutine writeList

end module LinkedList

!============================================================

program test
   use LinkedList
   implicit none
   character(len=*), parameter :: fmt = "( i0, 1x )"  ! Format for output
   integer, allocatable :: A(:)
   integer i

   A = [ 4, 4, 10, 4, 3, 5, 4 ]

   do i = 1, size( A )
      call push_back( A( i ) )         ! Add contents of A sequentially to back
   end do
   call writeList( fmt )

   call erase( 4 )                     ! Delete a particular value
   call writeList( fmt )

   do i = 1, 10                        ! Add numbers sequentially to front
      call push_front( i )
   end do
   call writeList( fmt )

end program test
```

```
4 4 10 4 3 5 4
10 3 5
10 9 8 7 6 5 4 3 2 1 10 3 5
```

16.3 Function Pointers

As well as pointers to variables we can have pointers to procedures. Just like pointers to variables, these *procedure pointers* can be used as aliases for the procedures to which they point and their big advantage over hard-coded procedure names is that they can be set and changed at run time.

Whereas we have a pointer-to-variable declaration:

    type, `pointer ::` *pointer_name*

for a procedure (function or subroutine) we would declare

    `procedure(`*prototype*`), pointer ::` *pointer_name*

with the prototype calling sequence being defined by an abstract interface (Section 13).

This is illustrated in the following calculator example. (Note the division symbol: here '|', because I have to avoid the shell uses of '/'.)

```
module Calculator
   implicit none
   integer, parameter :: dp = kind( 1.0d0 )      ! Double-precision

   abstract interface
      function BinaryOp( a, b )
         import dp                               ! Needed to bring dp into scope
         real(kind=dp) BinaryOp
         real(kind=dp) a, b
      end function BinaryOp
   end interface

contains

   real(kind=dp) function add( a, b )
      real(kind=dp) a, b
      add = a + b
   end function add

   real(kind=dp) function subtract( a, b )
      real(kind=dp) a, b
      subtract = a - b
   end function subtract

   real(kind=dp) function multiply( a, b )
      real(kind=dp) a, b
      multiply = a * b
   end function multiply

   real(kind=dp) function divide( a, b )
      real(kind=dp) a, b
      divide = a / b
   end function divide

end module Calculator

!============================================================

program main
   use Calculator
   implicit none
   real(kind=dp) a, b
   character op
   procedure(BinaryOp), pointer :: f             ! Declare a function pointer

   write( *, * ) "Enter the calculation as   num1 op num2   (with spaces)"
   write( *, * ) "op should be one of +, -, *, | "
   read( *, * ) a, op, b

   select case( op )                             ! Associate function pointer
      case( '+' );  f => add
      case( '-' );  f => subtract
      case( '*' );  f => multiply
      case( '|' );  f => divide
      case default;   write( *, * ) "Invalid operator";   stop
   end select

   write( *, "( a, g11.4 )" ) "Result is ", f( a, b )   ! Function pointer used as an alias

end program main
```

```
Enter the calculation as   num1 op num2   (with spaces)
op should be one of +, -, *, |
3.5 * 2.6
Result is   9.100
```

17. Passing Functions as Procedure Arguments

See Sample Programs G

Consider the following type of problem.

Integrate, numerically:

$$\int_a^b f(x)\,\mathrm{d}x$$

What we would like to do is have a single integration routine (e.g. by trapezium rule, mid-ordinate rule, Simpson's rule or other method of quadrature) and pass it an *arbitrary* function $f(x)$ (together with the bounds a and b and number of intervals).

To do this we have to be able to pass the information (name, return type, and, ideally, calling sequence) from an invoking routine to the integration procedure. Thus we need to be able *to pass a function itself as an argument*.

To be able to do this, both invoking and invoked routines need some sort of interface to provide a description of the passed procedure's form. At the minimum they require an *implicit* interface which will tell them whether it is a subroutine or function and, if the latter, the return type. An implicit interface can be achieved with either `external` (user procedure) or `intrinsic` (built-in procedure) statement or attribute. However, a full description of the passed procedure can also be achieved by an *explicit* interface, which defines the:

- nature (function or subroutine);
- return type (if a function);
- types of all arguments.

17.1 Implicit Interface

The minimum requirements are that the procedure to be passed is declared `external` (for a user procedure) or `intrinsic` (for a built-in) procedure, as in the following example of numerical integration (using the mid-ordinate rule).

Note that, in both `function integrate` and `program example` the procedure to be passed by argument is declared by

```
real, external :: f
```

signifying that it is an external function (not an intrinsic procedure), and returns a `real`.

Exactly the same implicit interface can be achieved by the separate type and specification statements:

```
real f
external f
```

or by

```
procedure(real) f
```

If a subroutine (which has no return type) were to be used instead then it would not need a type statement:

`external` *subroutine_name*

or

`procedure()`*subroutine_name*

If the procedure to be passed is built-in by the Fortran standard, e.g. exp, then it is declare *intrinsic* instead and does not need a type statement:

```
intrinsic exp
```

```
module UtilityModule
   implicit none

contains

   real function integrate( f, low, high, n )
      real, external :: f                      ! Implicit interface
      real, intent(in) :: low, high
      integer, intent(in) :: n
      integer i
      real dx

      integrate = 0.0
      dx = ( high - low ) / n
      do i = 1, n
         integrate = integrate + f( low + ( i - 0.5 ) * dx )
      end do
      integrate = integrate * dx
   end function integrate

end module UtilityModule

!========================================

real function f1( x )
   real, intent(in) :: x

   f1 = x ** 2

end function f1

!========================================

real function f2( x )
   real, intent(in) :: x

   f2 = exp( x )

end function f2

!========================================

program example
   use UtilityModule               ! Access to integration routine
   implicit none
   real, external :: f1, f2        ! Implicit interface
   real :: a = 0.0, b = 3.0
   integer :: n = 1000

   write( *, * ) "1st integral = ", integrate( f1, a, b, n )
   write( *, * ) "2nd integral = ", integrate( f2, a, b, n )

end program example
```

17.2 Explicit Interface

An explicit interface allows full type-checking (otherwise, it is a case of "hope the user knows what he is doing" when it comes to passing a procedure taking the requisite number and type of arguments). It also allows the option of passing a procedure via a procedure pointer rather than via its name.

An explicit interface may be provided by:

- having the required functions in a module, or by `use` of that module;
- an `interface` block (see Section 13);
- an `abstract interface` (defining the call sequence of the function) and `procedure` statement.

In our second version of the integration example, we use an abstract interface (to define the call sequence of a particular set of procedure), together with a procedure statement. Since the abstract interface is required in both invoking routine (`program example`) and invoked routine (`subroutine integrate`) we can avoid writing it twice by putting it in a module.

Note that the explicit interface is provided by a statement

```
   procedure(funcOneArgument) f
```

following the earlier definition of prototype `funcOneArgument` in the abstract interface:

```
   abstract interface
      real function funcOneArgument( x )
         real, intent(in) :: x
      end function funcOneArgument
   end interface
```

```
module UtilityModule
   implicit none

   abstract interface
      real function funcOneArgument( x )
         real, intent(in) :: x
      end function funcOneArgument
   end interface

contains

   real function integrate( f, low, high, n )
      procedure(funcOneArgument) f             ! Explicit interface
      real, intent(in) :: low, high
      integer, intent(in) :: n
      integer i
      real dx

      integrate = 0.0
      dx = ( high - low ) / n
      do i = 1, n
         integrate = integrate + f( low + ( i - 0.5 ) * dx )
      end do
      integrate = integrate * dx
   end function integrate

end module UtilityModule

!========================================

real function f1( x )
   real, intent(in) :: x

   f1 = x ** 2

end function f1

!========================================

real function f2( x )
   real, intent(in) :: x

   f2 = exp( x )

end function f2

!========================================

program example
   use UtilityModule                   ! Access to integration routine
   implicit none
   procedure(funcOneArgument) f1, f2   ! Explicit interface
   real :: a = 0.0, b = 3.0
   integer :: n = 1000

   write( *, * ) "1st integral = ", integrate( f1, a, b, n )
   write( *, * ) "2nd integral = ", integrate( f2, a, b, n )

end program example
```

17.3 Passing Function Pointers

Finally, we note that the procedure can be passed by a procedure pointer rather by its name. Maintaining the abstract interface to define the call sequence of the procedure, this requires minimal change to the above program; namely:

- In function `integrate` change the declaration to
  `procedure(funcOneArgument), pointer :: f`
  The pointer `f` is now an alias (or reference) when used within the routine.

- The main routine is now:

```
program example
   use UtilityModule
   implicit none
   procedure(funcOneArgument) f1, f2              ! External functions
   procedure(funcOneArgument), pointer :: ptr     ! Procedure pointer
   real :: a = 0.0, b = 3.0
   integer :: n = 1000

   ptr => f1
   write( *, * ) "1st integral = ", integrate( ptr, a, b, n )
   ptr => f2
   write( *, * ) "2nd integral = ", integrate( ptr, a, b, n )

end program example
```

Note that there is no need to declare `f1` and `f2` as external (that is implied by the `procedure` statement) and no need for them to have a target attribute.

18 Object-Oriented Fortran

See Sample Programs H

18.1 Terminology

Object-oriented programming (OOP) is a software paradigm where quantities are *objects* of particular *classes*, having their own *properties* (variables) and *methods* (functions). Languages like Java and C# are completely object-oriented – everything involves a class; languages like Fortran and C++ admit both *procedural* and *object-oriented* programming.

A class may be regarded as an enhanced form of user-defined type. Individual variables of that type are called *objects* of that class and are said to be *instantiated* when they are declared (and initialised). If desired, initialisation can be done by particular functions called *constructors* and many objects also have *destructors* which are called at the end of their life (e.g. to carry out deallocation and avoid memory leaks).

A precise definition of object-oriented programming is impossible, but several key notions are common:

Encapsulation – "Self-contained". The properties of an object can only be changed via a *public interface* consisting of carefully-specified member functions. Other data and function members are *private* and not accessible from outside the object. This allows the inner working of the function to be modified without affecting external code.

Inheritance – Classes may exist in a hierarchy, ranging from the general to the increasingly specialised. An analogy in real life would be animals – mammals – cats – tigers. There is a *base class* containing minimal properties and methods, with successive *derived classes* adding more properties and methods, and possibly *overriding* methods of the base class.

Polymorphism – "One interface, many forms". There are many aspects of this. The first aspect is *compile-time polymorphism*, consisting primarily of *overloaded* functions with the same *generic* name. For example, `abs(x)` may be called irrespective of whether `x` is `integer`, `real`, `complex`, ... The second aspect, *run-time polymorphism* allows one to access derived-class members via a base-class pointer or allocatable variable: the type of object may not be known until run-time. For example, with our zoological analogy we might use an animal pointer to point to a cat. In general, this would only allow us to access the properties of "cat" that are defined for "animal"; however, if some animal functions are designated as *virtual* (or *abstract*) then they form a template which is expected to be specially modified or overridden for a cat and they can be accessed by the animal-class pointer, with the particular specialisation detected at run-time.

The Fortran implementation of the object-oriented paradigm is summarised in the following table.

Object-oriented Feature	Fortran version
Class	Derived `type`
Method (or *member function*)	Type-bound procedure
Encapsulation	Modules; `public`, `private` or `protected` access
Inheritance	Type-extension.
Polymorphism	Compile-time: `generic` functions. Run-time: polymorphic variables; `class()` designator; `select type`; `abstract` types and `deferred` procedures

18.2 Example of a Fortran Class

In Fortran a typical model for each class is a distinct module which includes: the user-defined type, its type-bound procedures ("member functions") and related utility functions and operators. Thus:

```
module module_name
   type type_name
      variables
   contains
      type-bound procedures
   end type type_name

   interfaces to constructors and operators

contains
   constructors and destructors

   type-bound procedures

   utility functions

end module module_name
```

To give us something concrete to talk about, the following is an abbreviated fraction (or rational-number) class, together with a driver program to test it.

Fractions

$$\frac{p}{q}$$

are stored by their numerator, p, and denominator, q, in lowest integer form, with q positive. For brevity, the following class has only one defined numerical operation, namely addition:

$$\frac{a}{b}+\frac{c}{d}=\frac{ad+bc}{bd}$$

(available by both type-bound and non-type-bound procedures). However, it is easy to see how further operators could easily be added to this class and you are encouraged to do so.

```
module FractionClass
   implicit none

   ! Access arrangments
   private                                   ! Default arrangement; here (changed to) private
   public Fraction, operator( + ), add       ! ... except for limited public interface

   !*******************
   ! The actual class *
   !*******************
   type Fraction                             ! Class must be public in order to instantiate
      private                                ! ... Make data members private if desired
      integer p                              ! ... DON'T make private if to be type-extended
      integer q
   contains                                  ! List TYPE-BOUND procedures
      procedure    toString
      procedure :: plusEquals => pe          ! Note the optional renaming with =>
      final        finish                    ! FINALISER, aka "destructor"
   end type

   ! Constructors
   interface Fraction                        ! CONSTRUCTOR interface has same name as type
      procedure constructor_default
      procedure constructor_integer
      procedure constructor_text
   end interface Fraction

   ! Operators                               ! Associate (non-type-bound) procedures
   interface operator( + )
      procedure add
   end interface

contains

   !*****************************
   ! Constructors and Destructor *
```

```
   !*****************************

   function constructor_default( p, q ) result( this ) ! Construction from top and bottom
      integer, intent( in ) :: p, q
      type(Fraction) this

      this%p = p
      this%q = q

      call reduce( this%p, this%q )

   end function constructor_default

   !---------------------------------------------

   function constructor_integer( p ) result( this )   ! Construction from a whole number
      integer, intent( in ) :: p
      type(Fraction) this

      this%p = p
      this%q = 1

   end function constructor_integer

   !---------------------------------------------

   function constructor_text( text ) result( this )   ! Construction from character text
      character (len=*), intent( in ) :: text
      type(Fraction) this
      integer pos

      pos = scan( text, '/' )
      if ( pos > 0 ) then
         read( text( 1:pos-1 ), * ) this%p
         read( text( pos+1:  ), * ) this%q
      else
         read( text, * ) this%p
         this%q = 1
      end if

      call reduce( this%p, this%q )

   end function constructor_text

   !---------------------------------------------

   subroutine finish( this )                          ! Finaliser
      type(Fraction) this

!     write( *, * ) "Finaliser called"                ! Uncomment to see its effect

   end subroutine finish

   !---------------------------------------------

   !**********************
   ! Type-bound functions *
   !**********************

   function toString( this )                          ! Output the fraction as a string
      class(Fraction) this
      character(len=:), allocatable :: toString

      allocate( character(len=100) :: toString )

      if ( this%q == 1 ) then
         write( toString, "( i0 )" ) this%p
      else
         write( toString, "( i0, '/', i0 )" ) this%p, this%q
      end if

      toString = toString( 1:len_trim( toString ) )

   end function toString

   !---------------------------------------------

   subroutine pe( this, f )                           ! Add f to object
      class(Fraction) this
      type(Fraction), intent( in ) :: f
```

```
      this%p = this%p * f%q + this%q * f%p
      this%q = this%q * f%q

      call reduce( this%p, this%q )

   end subroutine pe

   !----------------------------------------------

   !*******************
   ! Utility functions *
   !*******************

   type(Fraction) function add( a, b )               ! Ordinary addition
      type(Fraction), intent(in) :: a, b

      add%p = a%p * b%q + a%q * b%p
      add%q = a%q * b%q

      call reduce( add%p, add%q )

   end function add

   !----------------------------------------------

   subroutine reduce( p, q )                          ! Write fraction in lowest terms
      integer, intent(inout) :: p, q
      integer h

      h = hcf( abs( p ), abs( q ) )                   ! Divide out the HCF
      p = p / h
      q = q / h

      if ( q < 0 ) then                               ! Ensure q is positive
         p = -p
         q = -q
      end if

   end subroutine reduce

   !----------------------------------------------

   recursive integer function hcf( a, b ) result( answer ) ! Highest common factor
      implicit none
      integer, intent(in) :: a, b

      if ( b == 0 ) then
         answer = a
      else
         answer = hcf( b, modulo( a, b ) )
      end if

   end function hcf

   !----------------------------------------------

end module FractionClass

!==========================================================

program testFractions
   use FractionClass
   implicit none
   type(Fraction) a, b, c, d, e

   a = Fraction( 2, 5  )                     ! uses constructor_default
   b = Fraction( 1 )                         ! uses constructor_integer
   c = Fraction( "4/12" )                    ! uses constructor_text
   write( *, * ) "a = ", a%toString()
   write( *, * ) "b = ", b%toString()
   write( *, * ) "c = ", c%toString()

   d = a + b                                 ! uses "add" via "+" symbol
   e = add( b, c )                           ! uses "add" directly
   write( *, * ) "a + b = ", d%toString()
   write( *, * ) "b + c = ", e%toString()

   call a%plusEquals( c )                    ! does the operation += ON the object a
   write( *, * ) "a -> a + c yields ", a%toString()

end program testFractions
```

```
a = 2/5
b = 1
c = 1/3
a + b = 7/5
b + c = 4/3
a -> a + c yields 11/15
```

We now dissect this example.

Module

A single module is dedicated to this class. It is made accessible where needed by a `use` statement:

```
use FractionClass
```

Using the Class

Turn to the main program first.

Variables of the class are declared and then initialised (in OOP terminology, *objects* are *instantiated*) by:

- normal type-declaration statements; e.g.
```
type(Fraction) a, b, c, d, e
```
- use of constructors; e.g.
```
a = Fraction( 2, 5  )
b = Fraction( 1 )
c = Fraction( "4/12" )
```

Note that there may be several constructors, corresponding to the various ways that a fraction might be expressed; here respectively:

– as the ratio of numbers *p* and *q*
– as the single number *p* (for a whole number)
– as a text string

The object is here output by using the type-bound procedure toString(); here, e.g.

```
a%toString()
```

Operations can be formed on object(s) using either type-bound procedures (e.g.

```
call a%plusEquals( c )
```

or non-type procedures:

```
e = add( b, c )
```

or equivalent operator syntax:

```
e = a + b
```

The Derived Type

Now turn to the module.

The fundamental part is derived type. This declares both data variables and type-bound procedures:

```
type Fraction
   private
   integer p
   integer q
contains
   procedure    toString
   procedure :: plusEquals => pe
   final        finish
end type
```

Note that procedures can optionally be renamed by the => operator.

Constructor(s)

In Fortran this is accomplished by:
- an interface with the same name as the user-defined type;
- one or more procedures that define individual constructors.

```
interface Fraction
   procedure constructor_default
   procedure constructor_integer
   procedure constructor_text
end interface Fraction
```

It is common to have more than one constructor, corresponding to the different ways that an object might be initialised.

Note that user-defined types usually have an implicit default *type-constructor*, which simply initialises the type elements in order. Here, it would be

```
Fraction( p, q )
```

However, this is not available here because:
- the data members `p` and `q` are declared `private`, so couldn't be accessed in this way;
- it is desired that the constructor do more than just initialise numerator and denominator; here, it also reduces them to their lowest form.

Destructor

In Fortran, this is called a *finaliser*. It is not often needed, but, where it is, the particular procedure used is indicated in the list of type-bound procedures.

```
final finish
```

The finaliser is only called if an object goes out of scope (e.g. by deallocation, or, for a local variable, at the end of the procedure in which it is declared) before the end of the program.

The main use of a finaliser is to recover memory that was dynamically allocated, avoiding a memory leak. As written here it will not actually be called.

Operators

Many operators may be declared by `interface` blocks listing the procedures for which the operator is a shorthand. e.g.

```
interface operator( + )
   procedure add
end interface
```

In this example,

```
c = a + b
```

would simply be shorthand for

```
c = add( a, b )
```

Type-Bound Procedures

Type-bound procedures ("*methods*") are applied directly to their associated object. The method to be used is listed amongst the procedures associated with that type; e.g. in the preceding example:

```
type Fraction
   ...
contains
   procedure toString
   procedure :: plusEquals => pe
   ...
end type
```

For the second of those listed procedures, the outside world uses the (public) name `plusEquals`, which is a renaming of the (private) internal routine `pe`.

They are called with the usual component selector for that object

object%method(*argument_list*)

However, the procedure itself receives one additional argument at the start – that of the object itself:

method(*this*, *argument_list*)
Moreover, when declaring the type of this first argument we must use the special form
`class(`*type_name*`) this`
instead of
`type(`*type_name*`) this`
`class` signifies a *polymorphic* type – the object referred to may be of that type … or any of its *type extensions* – see later.

For example, the main routine in our class example invokes our equivalent of the `+=` operator that is used in many languages[4]:

```
call a%plusEquals( c )
```

whereas the internal procedure (remembering the renaming `plusEquals => pe`) is declared as

```
subroutine pe( this, f )
   class( Fraction ) this
   type( Fraction ), intent( in ) :: f
```

This automatic passing of the object itself to a type-bound procedure can be overridden with the `nopass` attribute in the `procedure` statement within the `type`, but this is used only in specialised contexts (e.g., communicating with C or C++ routines).

Other Functions

It is rather common for the class to require non-type-bound functions to, as here, define operator functions for variables of that type (e.g. `add`) or do other necessary duties (`reduce()` and `hcf()`). The latter are often called *utility routines*. A natural place to put them is amongst the internal procedures of the same module.

Access

The principles of *encapsulation* require that access to variables or procedures from outside the class be limited to what is absolutely necessary – the *public interface*. Thus, outside of this module, only variables and procedures which have the `public` attribute are visible.

In Fortran, the default accessibility is `public` (we have nothing to hide!). Thus, if going along with privacy (which by no means all Fortran programmers do!) a common approach is to, as here, put a

```
private
```

statement amongst the type declarations. By default this makes all the contents of the module inaccessible from outside that module. Then we must selectively overrule this for a small list of components; e.g.

```
public Fraction, operator( + ), add
```

Note that the user type Fraction must be `public` or no such object could be instantiated. Putting it in this list, however, makes all its data variables `public`, so this has to be overruled once more for the properties; thus:

```
type Fraction
   private
   integer p
   integer q
contains
   ...
end type
```

As an alternative to declaring quantities `public` by a separate statement in list form we can also specify it individually as an attribute when declaring type; e.g.

```
type, public :: Fraction
   private
   integer p
   integer q
contains
   ...
end type
```

Making properties `private` is not, however, always possible, because they would be inaccessible to any *derived* (*type-extended*) class[5] – see later. Also some classes may be so trivial as to deem it not useful to make the data members private: they are just POD ("*plain old data*").

[4] This would be on my wish-list for Fortran, too!

18.3 Inheritance and Polymorphism

Run-time polymorphism is expressed in Fortran by:
- type extension (`type, extends(`*name*`)`)
- polymorphic variables (`class(`*basename*`)`)

18.3.1 Type Extension

Consider a user-defined type for a 2-d point:
```
   type point2d
      real x
      real y
   end type point2d
```

A natural extension of this is a 3-d coordinate, which adds a third coordinate, z, to the existing pair x, y:
```
   type, extends(point2d) :: point3d
      real z
   end type point3d
```

We say that the `point3d` *extends* or *inherits from* `point2d.` In the language of object-oriented programming, `point2d` and `point3d` are the *base* and *derived* (or *parent* and *child*) types, respectively. (It is unfortunate in this context that Fortran uses the term "derived type" to refer to any user-defined type, not just one that is inherited.)

If we now define a variable of type `point3d`:
```
   type(point3d) pt
```
then it has components `pt%x`, `pt%y` that it inherits from `point2d`, as well as an additional component `pt%z`. In fact, it also has a composite component of the parent type, `pt%point2d`, whilst the component `pt%x` can also (but less conveniently) be written `pt%point2d%x`.

Type extension can be continued to an arbitrary number of levels, giving a whole inheritance hierarchy.

18.3.2 Bound Procedures of Extended Types

As well as extra data fields, extended types may add additional bound procedures (member functions and subroutines). However, often we need to *override* procedures of the parent type to be more appropriate to the child type. For our coordinate example above we might like to define the radius by

$r = \sqrt{x^2 + y^2}$ in 2d

$r = \sqrt{x^2 + y^2 + z^2}$ in 3d

There are 2 main ways in which type-bound procedures can be overridden for ordinary variables. (We will see below that there are other ways for pointer or allocatable variables of polymorphic type.)

(1) Use the redirection operator => to redirect the common name (e.g. `radius`) to specific routines for each type.

(2) Use the `select type` construct for a polymorphic variable.

<u>Redirection</u>

In the following example, we redirect procedure `radius` to `r2d` or `r3d` for the two different types. Since those procedures themselves are private, the routine is always accessed by `p%radius()`, irrespective of whether `p` is `type(point2d)` or `type(point3d)`.

[5] C++ gets around this with its additional `protected` attribute, which allows access to any derived class. In Fortran, the `protected` attribute means read-only access: something completely different.

```
module Defs
   implicit none

   private
   public point2d, point3d

   type point2d
      real x
      real y
   contains
      procedure :: radius => r2d                ! Note the redirection for type point2d
   end type point2d

   type, extends(point2d) :: point3d            ! Extended type
      real z
   contains
      procedure :: radius => r3d                ! Note the redirection for type point3d
   end type point3d

contains

   !-----------------------------------

   real function r2d( this )
      class(point2d) this

      r2d = sqrt( this%x ** 2 + this%y ** 2 )

   end function r2d

   !-----------------------------------

   real function r3d( this )
      class(point3d) this

      r3d = sqrt( this%x ** 2 + this%y ** 2 + this%z ** 2 )

   end function r3d

   !-----------------------------------

end module Defs

!============================================================

program main
   use Defs
   implicit none
   type(point2d) :: p2d = point2d( 3, 4    )
   type(point3d) :: p3d = point3d( 3, 4, 5 )
   type(point2d) :: projection

   write( *, * ) "2-d radius is ", p2d%radius()
   write( *, * ) "3-d radius is ", p3d%radius()

   projection = p3d%point2d                     ! type is point2d
   write( *, * ) "Projected radius is ", projection%radius()

end program main
```

Using `Select Type`

The `class` keyword means that a single routine can be defined for the *polymorphic* type `class(point2d)` which applies to variables of `type(point2d)` *and any extension*. What is done for any particular extended type is specified by the select type construct, which has the general form

`select type (`*polymorphic_type*`)`
    `type is (`*type1*`)`
        instructions for type1
    `[type is (`*type2*`)`
        instructions for type2]
    `[class is (` *type3* `)`
        instructions for type3`]`
    `[class default`
      instructions for anything else`]`
`end select`

The latter groups are optional.

For example, the coordinate extension in the above example can be written as below.

```
module Defs
   implicit none

   private
   public point2d, point3d

   type point2d
      real x
      real y
   contains
      procedure radius                 ! Single function for all extensions
   end type point2d

   type, extends(point2d) :: point3d
      real z
   end type point3d

contains

   !-----------------------------------

   real function radius( this )
      class(point2d) this              ! Allows point2d ... or any EXTENSION

      select type( this )              ! Run-time selection of type
         type is ( point2d )
             radius = sqrt( this%x ** 2 + this%y ** 2 )
         type is ( point3d )
             radius = sqrt( this%x ** 2 + this%y ** 2 + this%z ** 2 )
         class default
             radius = -1.0
      end select

   end function radius

   !-----------------------------------

end module Defs

!============================================================

program main
   use Defs
   implicit none
   type(point2d) :: p2d = point2d( 3, 4    )
   type(point3d) :: p3d = point3d( 3, 4, 5 )
   type(point2d) :: projection

   write( *, * ) "2-d radius is ", p2d%radius()
   write( *, * ) "3-d radius is ", p3d%radius()

   projection = p3d%point2d                      ! type is point2d
   write( *, * ) "Projected radius is ", projection%radius()

end program main
```

18.3.3 Polymorphic Variables

In general a variable declared as

`class(`*basename*`)` *variable*

is called a *polymorphic* variable. It can be used to refer to any variable of type *basename* (called the *declared type*) or any extended type (called the *dynamic type*).

Because its type is determined (and may change) at run time it needs to be able to acquire its dynamic type from the variable to which it refers, and hence must be either:

- a pointer,
- allocatable, or
- a dummy argument of a procedure,

from which it acquires its type from its target, its allocation or an actual argument, respectively. We have already seen the last usage in the context of type-bound procedures. (In other languages, only a pointer to the base type is polymorphic; Fortran is blessed with two very powerful alternatives.)

Polymorphic variables used in this way depend a great deal on the `select type` construct to carry out the correct operations for the dynamic type at run-time.

There is one supreme and powerful level of abstraction, a variable of *unlimited polymorphic type*, denoted

```
class(*)
```

which can be used as a base for any type of variable. Metcalf, Reid and Cohen's 2018 book gives a superb example of the use of `class(*)` in a linked list, allowing nodal values to be of any type.

In the following example, variable `p` is an allocatable variable of unlimited polymorphic type, and is allocated in turn to each of the intrinsic types: `integer`, `real`, `complex`, `character`, `logical`. (In the next section we shall see an example with the `pointer` attribute instead.) In the called procedure a similar variable is declared as a dummy argument.

*** *WARNING* ****:* in the example below, allocation (and reallocation) is automatic by assignment. This is quite a recent addition to the language and some compilers have not yet implemented this in full. For those compilers, one could replace, e.g.,

```
p = 3;   call output( p )
```

by an explicit allocation (with the `source` parameter):

```
allocate( p, source = 3 );   call output( p );   deallocate( p )
```

which is nothing like as convenient!

```
program main
   class(*), allocatable :: p      ! Unlimited polymorphic variable, with allocatable attribute

   ! Note: automatic allocation or deallocation by assignment
   p = 3;                        call output( p )
   p = 3.14159;                  call output( p )
   p = cmplx( 1.0, 2.0 );        call output( p )
   p = "Mathematical pi";        call output( p )
   p = .true.;                   call output( p )

contains

   subroutine output( var )
      implicit none
      class(*) var                 ! Unlimited polymorphic variable, with argument association

      select type( var )
         type is ( integer )
            write( *, * ) "Integer: ", var
         type is ( real )
            write( *, * ) "Real: ", var
         type is ( complex )
            write( *, * ) "Complex: ", var
         type is ( logical )
            write( *, * ) "Logical: ", var
         type is ( character(len=*) )
            write( *, * ) "Character: ", var
         class default
            write( *, * ) "Variable type not recognised"
      end select

   end subroutine output

end program main
```

```
 Integer:  3
 Real:    3.1415901
 Complex:  (1.0000000, 2.0000000)
 Character: Mathematical pi
 Logical:  T
```

18.4 Abstract Types

Many languages (notably C++, with its input/output streams) use type extension and polymorphic variables to assemble a hierarchy of inherited types. Since an allocatable or pointer polymorphic variable of the base type can be made to refer to any extended type at run-time and the appropriate bound procedures can be called provided there is a prototype of the required name bound to the base type, it is quite common to make the base type *abstract*, which merely defines the calling sequence for named procedures, without defining them. The base type is given the `abstract` attribute and its procedures are given the `deferred` attribute, with their calling sequences defined by an `abstract interface`. (In other languages, deferred procedures are called *virtual functions*.) Extended types must individually `override` the procedure for their own particular needs.

Since an abstract type is incomplete (its procedures haven't been fully defined) no ordinary variable of this type may be instantiated. However, a pointer or allocatable variable of this type *can* be declared and pointed, or allocated, to a variable of any extended type at run time.

In the following example there is an `abstract` base type `point`, which is type-extended to `point2d` and then further to `point3d`. (Note that the abstract interface must use the `import` keyword to bring the definition of `type(point)` into its scope). In program main a base-type pointer is successively pointed to variables of its extended types `point2d` and `point3d`.

```
module Defs
   implicit none

   private
   public point, point2d, point3d

   type, abstract :: point                        ! Abstract parent type
   contains
      procedure(func), deferred :: radius         ! Prototype for func supplied by abstract interface
   end type point

   abstract interface
      real function func( this )
         import point                             ! Need to import type information into interface
         class(point) this
      end function func
   end interface


   type, extends(point) :: point2d                ! Child type
      real x, y
   contains
      procedure :: radius => r2d
   end type point2d

   type, extends(point2d) :: point3d              ! Grandchild type!
      real z
   contains
      procedure :: radius => r3d
   end type point3d

contains

   !-----------------------------------

   real function r2d( this )
      class(point2d) this

      r2d = sqrt( this%x ** 2 + this%y ** 2 )

   end function r2d

   !-----------------------------------

   real function r3d( this )
      class(point3d) this

      r3d = sqrt( this%x ** 2 + this%y ** 2 + this%z ** 2 )

   end function r3d

   !-----------------------------------

end module Defs

!============================================================

program main
```

```
   use Defs
   implicit none
   class(point), pointer :: ptr                     ! Polymorphic base type pointer
   type(point2d), target :: p2d  = point2d( 3, 4    )
   type(point3d), target :: p3d  = point3d( 3, 4, 5 )

   ptr => p2d;   write( *, * ) "2-d radius is ", ptr%radius()   ! Points to variable of type point2d
   ptr => p3d;   write( *, * ) "3-d radius is ", ptr%radius()   ! Points to variable of type point3d

end program main
```

```
2-d radius is    5.0000000
3-d radius is    7.0710678
```

As well as using a base-type pointer (similar to C++), Fortran can also use an allocatable variable of the base type, with allocation (and reallocation) conveniently done by assignment *(*** WARNING *** needs a very up-to-date compiler)*. For example, the main program in the above example can be changed to the following.

```
program main
   use Defs
   implicit none
   class(point), allocatable :: pt                  ! Allocatable polymorphic variable of base type

   pt  = point2d( 3, 4    );                        ! Note automatic allocation
   write( *, * ) "2-d radius is ", pt%radius()

   pt = point3d( 3, 4, 5 )                          ! Note automatic reallocation
   write( *, * ) "3-d radius is ", pt%radius()

end program main
```