FREE eBook

LEARNING Fortran

Free unaffiliated eBook created from
Stack Overflow contributors.

#fortran

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: fortran

It is an unofficial and free Fortran ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Fortran.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Fortran

Remarks

Fortran is a language used extensively in the scientific community due to its suitability for numerical computation. Particularly attractive is its intuitive array notation, which makes writing fast vectorised computations easy.

Despite its age, Fortran is still actively developed, with numerous implementations, including GNU, Intel, PGI and Cray.

Versions

Version	Note	Release
FORTRAN 66	First standardization by ASA (now ANSI)	1966-03-07
FORTRAN 77	Fixed Form, Historic	1978-04-15
Fortran 90	Free Form, ISO Standard, Array operations	1991-06-15
Fortran 95	Pure and Elemental Procedures	1997-06-15
Fortran 2003	Object Oriented Programming	2004-04-04
Fortran 2008	Co-Arrays	2010-09-10

Examples

Installation or Setup

Fortran is a language which can be compiled using compilers supplied by many vendors. Different compilers are available for different hardware platforms and operating systems. Some compilers are free software, some can be used free of charge and some require the purchase of a licence.

The most common free Fortran compiler is GNU Fortran or gfortran. The source code is available from GNU as a part of GCC, the GNU compiler collection. Binaries for many operating systems are available at https://gcc.gnu.org/wiki/GFortranBinaries. Linux distributions often contain gfortran in their package manager.

Further compilers are available for example:

- EKOPath by PathScale
- LLVM (backend via DragonEgg)
- Oracle Developer Studio

- Absoft Fortran Compiler
- Intel Fortran Compiler
- NAG Fortran Compiler
- PGI Compilers

On HPC-Systems there are often specialized compilers available by the system provider as for example the IBM or Cray compilers.

All these compilers support the Fortran 95 standard. An overview on the Fortran 2003 status and the Fortran 2008 status by various compilers is offered by the ACM Fortran Forum and available in the Fortran Wiki.

Hello, world

Any Fortran program has to include `end` as last statement. Therefore, the simplest Fortran program looks like this:

```
end
```

Here are some examples of "hello, world" programs:

```
print *, "Hello, world"
end
```

With `write` statement:

```
write(*,*) "Hello, world"
end
```

For clarity it is now common to use the `program` statement to start a program and give it a name. The `end` statement can then refer to this name to make it obvious what it is referring to, and let the compiler check the code for correctness. Further, all Fortran programs should include an `implicit none` statement. Thus, a minimal Fortran program actually should look as follows:

```
program hello
  implicit none
  write(*,*) 'Hello world!'
end program hello
```

The next logical step from this point is how to see the result of the hello world program. This section shows how to achieve that in a linux like environment. We assume that you have some basic notions of shell commands, mainly you know how to get to the shell terminal. We also assume that you have already setup your `fortran` environment. Using your preferred text editor (notepad, notepad++, vi, vim, emacs, gedit, kate, etc.), save the hello program above (copy and paste) in a file named `hello.f90` in your home directory. `hello.f90` is your source file. Then go to the command line and navigate to the directory(home directory?) where you saved your source file, then type the following command:

```
>gfortran –o hello hello.f90
```

You just created your hello world executable program. In technical terms, you just compiled your program. To run it, type the following command:

```
>./hello
```

You should see the following line printed on your shell terminal.

```
> Hello world!
```

Congratulations, you just wrote, compiled and ran the "Hello World" program.

Quadratic equation

Today Fortran is mainly used for numerical computation. This very simple example illustrates the basic program structure to solve quadratic equations:

```
program quadratic
  !a comment

  !should be present in every separate program unit
  implicit none

  real :: a, b, c
  real :: discriminant
  real :: x1, x2

  print *, "Enter the quadratic equation coefficients a, b and c:"
  read *, a, b, c

  discriminant = b**2 - 4*a*c

  if ( discriminant>0 ) then

    x1 = ( -b + sqrt(discriminant)) / (2 * a)
    x2 = ( -b - sqrt(discriminant)) / (2 * a)
    print *, "Real roots:"
    print *, x1, x2

    ! Comparison of floating point numbers for equality is often not recommended.
    ! Here, it serves the purpose of illustrating the "else if" construct.
  else if ( discriminant==0 ) then

    x1 = - b / (2 * a)
    print *, "Real root:"
    print *, x1
  else

    print *, "No real roots."
  end if
end program quadratic
```

Case insensitivity

Uppercase and lowercase letters of the alphabet are equivalent in the Fortran character set. In other words, Fortran is *case insensitive*. This behavior is in contrast with case-sensitive languages, such as C++ and many others.

As a consequence, the variables `a` and `A` are the same variable. In principle one could write a program as follows

```
pROgrAm MYproGRaM
..
enD mYPrOgrAM
```

It's to the good programmer to avoid such ugly choices.

Read Getting started with Fortran online: https://riptutorial.com/fortran/topic/904/getting-started-with-fortran

Chapter 2: Arrays

Examples

Basic notation

Any type can be declared as an array using either the *dimension* attribute or by just indicating directly the `dimension`(s) of the array:

```
! One dimensional array with 4 elements
integer, dimension(4) :: foo

! Two dimensional array with 4 rows and 2 columns
real, dimension(4, 2) :: bar

! Three dimensional array
type(mytype), dimension(6, 7, 8) :: myarray

! Same as above without using the dimension keyword
integer :: foo2(4)
real :: bar2(4, 2)
type(mytype) :: myarray2(6, 7, 8)
```

The latter way of declaring multidimensional array, allows the declaration of same-type different-rank/dimensions arrays in one line, as follows

```
real :: pencil(5), plate(3,-2:4), cuboid(0:3,-10:5,6)
```

The maximum rank (number of dimensions) allowed is 15 in Fortran 2008 standard and was 7 before.

Fortran stores arrays in *column-major* order. That is, the elements of `bar` are stored in memory as follows:

```
bar(1, 1), bar(2, 1), bar(3, 1), bar(4, 1), bar(1, 2), bar(2, 2), ...
```

In Fortran, array numbering starts at **1** by default, in contrast to C which starts at **0**. In fact, in Fortran, you can specify the upper and lower bounds for each dimension explicitly:

```
integer, dimension(7:12, -3:-1) :: geese
```

This declares an array of shape `(6, 3)`, whose first element is `geese(7, -3)`.

Lower and upper bounds along the 2 (or more) dimensions can be accessed by the intrinsic functions `ubound` and `lbound`. Indeed `lbound(geese,2)` would return `-3`, whereas `ubound(geese,1)` would return `12`.

Size of an array can be accessed by intrinsic function `size`. For example, `size(geese, dim = 1)`

returns the size of first dimension which is 6.

Allocatable arrays

Arrays can have the *allocatable* attribute:

```
! One dimensional allocatable array
integer, dimension(:), allocatable :: foo
! Two dimensional allocatable array
real, dimension(:,:), allocatable :: bar
```

This declares the variable but does not allocate any space for it.

```
! We can specify the bounds as usual
allocate(foo(3:5))

! It is an error to allocate an array twice
! so check it has not been allocated first
if (.not. allocated(foo)) then
  allocate(bar(10, 2))
end if
```

Once a variable is no longer needed, it can be *deallocated*:

```
deallocate(foo)
```

If for some reason an `allocate` statement fails, the program will stop. This can be prevented if the status is checked via the `stat` keyword:

```
real, dimension(:), allocatable :: geese
integer :: status

allocate(geese(17), stat=status)
if (stat /= 0) then
  print*, "Something went wrong trying to allocate 'geese'"
  stop 1
end if
```

The `deallocate` statement has `stat` keyword too:

```
deallocate (geese, stat=status)
```

`status` is an integer variable whose value is 0 if the allocation or deallocation was successful.

Array constructors

A rank-1 array value can be created using an *array constructor*, with the syntax

```
(/ ... /)
[ ... ]
```

The form `[...]` was introduced in Fortran 2003 and is generally regarded as clearer to read, especially in complex expressions. This form is used exclusively in this example.

The values featuring in an array constructor may be scalar values, array values, or implied-do loops.

The type and type parameters of the constructed array match those of the values in the array constructor

```
[1, 2, 3]      ! A rank-1 length-3 array of default integer type
[1., 2., 3.]   ! A rank-1 length-3 array of default real type
["A", "B"]     ! A rank-1 length-2 array of default character type

integer, parameter :: A = [2, 4]
[1, A, 3]      ! A rank-1 length-4 array of default integer type, with A's elements

integer i
[1, (i, i=2, 5), 6]  ! A rank-1 length-6 array of default integer type with an implied-do
```

In the forms above, all the values given must be of the same type and type parameter. Mixing types, or type parameters, is not allowed. The following examples **are not valid**

```
[1, 2.]      ! INVALID: Mixing integer and default real
[1e0, 2d0]   ! INVALID: Mixing default real and double precision
[1., 2._dp]  ! INVALID: Allowed only if kind `dp` corresponds to default real
["Hello", "Frederick"]  ! INVALID: Different length parameters
```

To construct an array using different types, a type specification for the array shall be given

```
[integer :: 1, 2., 3d0]    ! A default integer array
[real(dp) :: 1, 2, 3._sp]  ! A real(dp) array
[character(len=9) :: "Hello", "Frederick"]  ! A length-2 array of length-9 characters
```

This latter form for character arrays is especially convenient to avoid space padding, such as the alternative

```
["Hello    ", "Frederick"]  ! A length-2 array of length-9 characters
```

The size of an array named constant may be implied by the array constructor used to set its value

```
integer, parameter :: ids(*) = [1, 2, 3, 4]
```

and for length-parameterized types the length parameter may be assumed

```
character(len=*), parameter :: names(*) = [character(3) :: "Me", "You", "Her"]
```

The type specification is also required in the construction of zero-length arrays. From

```
[ ] ! Not a valid array constructor
```

the type and type parameters cannot be determined from the non-existing value set. To create a zero-length default integer array:

```
[integer :: ]
```

Array constructors construct only rank-1 arrays. At times, such as in setting the value of a named constant, higher rank arrays are also required in an expression. Higher rank arrays can be taken from the result of `reshape` with a constructed rank-1 array

```
integer, parameter :: multi_rank_ids(2,2) = RESHAPE([1,2,3,4], shape=[2,2])
```

In an array constructor the values of the array in element order with any arrays in the value list being as though the individual elemets were given themselves in array element order. Thus, the earlier example

```
integer, parameter :: A = [2, 4]
[1, A, 3]      ! A rank-1 length-4 array of default integer type, with A's elements
```

is equivalent to

```
[1, 2, 4, 3]   ! With the array written out in array element order
```

Generally the values in the constructor may be arbitrary expressions, including nested array constructors. For such an array constructor to meet certain conditions, such as being a constant or specification expression, restrictions apply to constituent values.

Although not an array constructor, certain array values may also be conveniently created using the `spread` intrinsic function. For example

```
[(0, i=1,10)]  ! An array with 10 default integers each of value 0
```

is also the result of the function reference

```
SPREAD(0, 1, 10)
```

Array nature specification: rank and shape

The `dimension` attribute on an object specifies that that object is an array. There are, in Fortran 2008, five array natures:[1]

- explicit shape
- assumed shape
- assumed size
- deferred shape
- implied shape

Take the three rank-1 arrays[2]

```
integer a, b, c
dimension(5) a    ! Explicit shape (default lower bound 1), extent 5
dimension(:) b    ! Assumed or deferred shape
dimension(*) c    ! Assumed size or implied shape array
```

With these it can be seen that further context is required to determine fully the nature of an array.

Explicit shape

An explicit shape array is always the shape of its declaration. Unless the array is declared as local to a subprogram or `block` construct, the bounds defining shape must be constant expressions. In other cases, an explicit shape array may be an automatic object, using extents which may vary on each invocation of a subprogram or `block`.

```
subroutine sub(n)
  integer, intent(in) :: n
  integer a(5)   ! A local explicit shape array with constant bound
  integer b(n)   ! A local explicit shape array, automatic object
end subroutine
```

Assumed shape

An assumed shape array is a dummy argument without the `allocatable` or `pointer` attribute. Such an array takes its shape from the actual argument with which it is associated.

```
integer a(5), b(10)
call sub(a)   ! In this call the dummy argument is like x(5)
call sub(b)   ! In this call the dummy argument is like x(10)

contains

  subroutine sub(x)
    integer x(:)    ! Assumed shape dummy argument
  end subroutine sub

end
```

When a dummy argument has assumed shape the scope referencing the procedure must have an explicit interface available for that procedure.

Assumed size

An assumed size array is a dummy argument which has its size assumed from its actual argument.

```
subroutine sub(x)
  integer x(*)   ! Assumed size array
end subroutine
```

Assumed size arrays behave very differently from assumed shape arrays and these differences are documented elsewhere.

Deferred shape

A deferred shape array is an array which has the `allocatable` or `pointer` attribute. The shape of such an array is determined by its allocation or pointer assignment.

```
integer, allocatable :: a(:)
integer, pointer :: b(:)
```

Implied shape

An implied shape array is a named constant which takes its shape from the expression used to establish its value

```
integer, parameter :: a(*) = [1,2,3,4]
```

The implications of these array declarations on dummy arguments are to be documented elsewhere.

---

[1]A Technical Specification extending Fortran 2008 adds a sixth array nature: assumed rank. This is not covered here.

[2] These can equivalently be written as

```
integer, dimension(5) :: a
integer, dimension(:) :: b
integer, dimension(*) :: c
```

or

```
integer a(5)
integer b(:)
integer c(*)
```

Whole arrays, array elements and array sections

Consider the array declared as

```
real x(10)
```

Then we have three aspects of interest:

1. The whole array `x`;
2. Array elements, like `x(1)`;

3. Array sections, like `x(2:6)`.

Whole arrays

In most cases the whole array `x` refers to all of the elements of the array as a single entity. It may appear in executable statements such as `print *, SUM(x)`, `print *, SIZE(x)` or `x=1`.

A whole array may reference arrays which aren't explicitly shaped (such as `x` above):

```
function f(y)
  real, intent(out) :: y(:)
  real, allocatable :: z(:)

  y = 1.         ! Intrinsic assignment for the whole array
  z = [1., 2.,]  ! Intrinsic assignment for the whole array, invoking allocation
end function
```

An assumed-size array may also appear as a whole array, but in limited circumstances only (to be documented elsewhere).

Array elements

An array element is referred to be giving integer indexes, one for each rank of the array, denoting the location in the whole array:

```
real x(5,2)
x(1,1) = 0.2
x(2,4) = 0.3
```

An array element is a scalar.

Array sections

An array section is a reference to a number of elements (perhaps just one) of a whole array, using a syntax involving colons:

```
real x(5,2)
x(:,1) = 0.          ! Referring to x(1,1), x(2,1), x(3,1), x(4,1) and x(5,1)
x(2,:) = 0.          ! Referring to x(2,1), x(2,2)
x(2:4,1) = 0.        ! Referring to x(2,1), x(3,1) and x(4,1)
x(2:3,1:2) = 0.      ! Referring to x(2,1), x(3,1), x(2,2) and x(3,2)
x(1:1,1) = 0.        ! Referring to x(1,1)
x([1,3,5],2) = 0.    ! Referring to x(1,2), x(3,2) and x(5,2)
```

The final form above uses a *vector subscript*. This is subject to a number of restrictions beyond other array sections.

Each array section is itself an array, even when just one element is referenced. That is `x(1:1,1)` is an array of rank 1 and `x(1:1,1:1)` is an array of rank 2.

Array sections do not in general have an attribute of the whole array. In particular, where

```
real, allocatable :: x(:)
x = [1,2,3]     ! x is allocated as part of the assignment
x = [1,2,3,4]   ! x is dealloacted then allocated to a new shape in the assignment
```

the assignment

```
x(:) = [1,2,3,4,5]   ! This is bad when x isn't the same shape as the right-hand side
```

is not allowed: `x(:)`, although an array section with all elements of `x`, is not an allocatable array.

```
x(:) = [5,6,7,8]
```

is fine when `x` is of the shape of the right-hand side.

Array components of arrays

```
type t
   real y(5)
end type t

type(t) x(2)
```

We may also refer to whole arrays, array elements and array sections in more complicated settings.

From the above, `x` is a whole array. We also have

```
x(1)%y        ! A whole array
x(1)%y(1)     ! An array element
x%y(1)        ! An array section
x(1)%y(:)     ! An array section
x([1,2]%y(1)  ! An array section
x(1)%y(1:1)   ! An array section
```

In such cases we are not allowed to have more than one part of the reference consisting of an array of rank 1. The following, for example, are not allowed

```
x%y             ! Both the x and y parts are arrays
x(1:1)%y(1:1)   ! Recall that each part is still an array section
```

Array operations

Due to its computational goals, mathematical operations on arrays are straight forward in Fortran.

Addition and subtraction

Operations on arrays of the same shape and size are very similar to matrix algebra. Instead of running through all the indices with loops, one can write addition (and subtraction):

```
real, dimension(2,3) :: A, B, C
real, dimension(5,6,3) :: D
A    = 3.    ! Assigning single value to the whole array
B    = 5.    ! Equivalent writing for assignment
C    = A + B ! All elements of C now have value 8.
D    = A + B ! Compiler will raise an error. The shapes and dimensions are not the same
```

Arrays from slicing are also valid:

```
integer :: i, j
real, dimension(3,2) :: Mat = 0.
real, dimension(3)   :: Vec1 = 0., Vec2 = 0., Vec3 = 0.
i = 0
j = 0
do i = 1,3
  do j = 1,2
    Mat(i,j) = i+j
  enddo
enddo
Vec1 = Mat(:,1)
Vec2 = Mat(:,2)
Vec3 = Mat(1:2,1) + Mat(2:3,2)
```

Function

In the same way, most intrinsic functions can be used implicitly assuming component-wise operation (though this is not systematic):

```
real, dimension(2) :: A, B
A(1) = 6
A(2) = 44 ! Random values
B    = sin(A) ! Identical to B(1) = sin(6), B(2) = sin(44).
```

Multiplication and division

Care must be taken with product and division: intrinsic operations using `*` and `/` symbols are element-wise:

```
real, dimension(2) :: A, B, C
A(1) = 2
A(2) = 4
B(1) = 1
B(2) = 3
C = A*B ! Returns C(1) = 2*1 and C(2) = 4*3
```

This must not be mistaken with matrix operations (see below).

Matrix operations

Matrix operations are intrinsic procedures. For example, the matrix product of the arrays of the previous section is written as follows:

```
real, dimension(2,1) :: A, B
real, dimension(1,1) :: C
A(1) = 2
A(2) = 4
B(1) = 1
B(2) = 3
C = matmul(transpose(A),B) ! Returns the scalar product of vectors A and B
```

Complex operations allow encapsulated of functions by creating temporary arrays. While allowed by some compilers and compilation options, this is not recommanded. For example, a product including a matrix transpose can be written:

```
real, dimension(3,3) :: A, B, C
A(:) = 4
B(:) = 5
C = matmul(transpose(A),matmul(B,matmul(A,transpose(B)))) ! Equivalent to A^t.B.A.B^T
```

Advanced array sections: subscript triplets and vector subscripts

As mentioned in another example a subset of the elements of an array, called an array section, may be referenced. From that example we may have

```
real x(10)
x(:)   = 0.
x(2:6) = 1.
x(3:4) = [3., 5.]
```

Array sections may be more general than this, though. They may take the form of subscript triplets or vector subscripts.

Subscript triplets

A subscript triple takes the form `[bound1]:[bound2][:stride]`. For example

```
real x(10)
x(1:10) = ...   ! Elements x(1), x(2), ..., x(10)
x(1:) = ...     ! The omitted second bound is equivalent to the upper, same as above
x(:10) = ...    ! The omitted first bound is equivalent to the lower, same as above
x(1:6:2) = ...  ! Elements x(1), x(3), x(5)
x(5:1) = ...    ! No elements: the lower bound is greater than the upper
x(5:1:-1) = ... ! Elements x(5), x(4), x(3), x(2), x(1)
x(::3) = ...    ! Elements x(1), x(4), x(7), x(10), assuming omitted bounds
x(::-3) = ...   ! No elements: the bounds are assumed with the first the lower, negative
stride
```

When a stride (which must not be zero) is specified, the sequence of elements begins with the first bound specified. If the stride is positive (resp. negative) the selected elements following a sequence incremented (resp. decremented) by the stride until the last element not larger (resp.

smaller) than the second bound is taken. If the stride is omitted it is treated as being one.

If the first bound is larger than the second bound, and the stride is positive, no elements are specified. If the first bound is smaller than the second bound, and the stride is negative, no elements are specified.

It should be noted that `x(10:1:-1)` is not the same as `x(1:10:1)` even though each element of `x` appears in both cases.

Vector subscripts

A vector subscript is a rank-1 integer array. This designates a sequence of elements corresponding to the values of the array.

```
real x(10)
integer i
x([1,6,4]) = ...     ! Elements x(1), x(6), x(4)
x([(i,i=2,4)]) = ... ! Elements x(2), x(3) and x(4)
print*, x([2,5,2])   ! Elements x(2), x(5) and x(2)
```

An array section with a vector subscript is restricted in how it may be used:

- it may not be argument associated with a dummy argument which is defined in the procedure;
- it may not be the target in a pointer assignment statement;
- it may not be an internal file in a data transfer statement.

Further, such an array section may not appear in a statement which involves its definition when the same element is selected twice. From above:

```
print*, x([2,5,2])   ! Elements x(2), x(5) and x(2) are printed
x([2,5,2]) = 1.      ! Not permitted: x(2) appears twice in this definition
```

Higher rank array sections

```
real x(5,2)
print*, x(::2,2:1:-1)  ! Elements x(1,2), x(3,2), x(5,2), x(1,1), x(3,1), x(5,1)
```

Read Arrays online: https://riptutorial.com/fortran/topic/996/arrays

Chapter 3: C interoperability

Examples

Calling C from Fortran

Fortran 2003 introduced language features which can guarantee interoperability between C and Fortran (and to more languages by using C as an intermediary). These features are mostly accessed through the intrinsic module `iso_c_binding`:

```
use, intrinsic :: iso_c_binding
```

The `intrinsic` keyword here ensures the correct module is used, and not a user created module of the same name.

`iso_c_binding` gives access to *interoperable* kind type parameters:

```
integer(c_int) :: foo    ! equivalent of 'int foo' in C
real(c_float) :: bar     ! equivalent of 'float bar' in C
```

Use of C kind type parameters guarantees that the data can be transferred between C and Fortran programs.

Interoperability of C char and Fortran characters is probably a topic for itself and so not discussed here

---

To actually call a C function from Fortran, first the interface must be declared. This is essentially equivalent to the C function prototype, and lets the compiler know about the number and type of the arguments, etc. The `bind` attribute is used to tell the compiler the name of the function in C, which may be different to the Fortran name.

geese.h

```
// Count how many geese are in a given flock
int howManyGeese(int flock);
```

geese.f90

```
! Interface to C routine
interface
  integer(c_int) function how_many_geese(flock_num) bind(C, 'howManyGeese')
    ! Interface blocks don't know about their context,
    ! so we need to use iso_c_binding to get c_int definition
    use, intrinsic :: iso_c_binding, only : c_int
    integer(c_int) :: flock_num
  end function how_many_geese
end interface
```

The Fortran program needs to be linked against the C library (*compiler dependent, include here?*) that includes the implementation of `howManyGeese()`, and then `how_many_geese()` can be called from Fortran.

C structs in Fortran

The `bind` attribute can also be applied to derived types:

geese.h

```
struct Goose {
   int flock;
   float buoyancy;
}

struct Goose goose_c;
```

geese.f90

```
use, intrinsic :: iso_c_binding, only : c_int, c_float

type, bind(C) :: goose_t
  integer(c_int) :: flock
  real(c_float) :: buoyancy
end type goose_t

type(goose_t) :: goose_f
```

Data can now be transferred between `goose_c` and `goose_f`. C routines which take arguments of type `Goose` can be called from Fortran with `type(goose_t)`.

Read C interoperability online: https://riptutorial.com/fortran/topic/2184/c-interoperability

Chapter 4: Data Types

Examples

Intrinsic types

The following are data types *intrinsic* to Fortran:

```
integer
real
character
complex
logical
```

`integer`, `real` and `complex` are numeric types.

`character` is a type used to store character strings.

`logical` is used to store binary values `.true.` or `.false.`.

All numeric and logical intrinsic types are parametrized using kinds.

```
integer(kind=specific_kind)
```

or just

```
integer(specific_kind)
```

where `specific_kind` is an integer named constant.

Character variables, as well as having a kind parameter, also have a length parameter:

```
character char
```

declares `char` to be a length-1 character variable of default kind, whereas

```
character(len=len) name
```

declares `name` to be a character variable of default kind and length `len`. The kind can also be specified

```
character(len=len, kind=specific_kind) name
character(kind=specific_kind) char
```

declares `name` to be a character of kind `kind` and length `len`. `char` is a length-1 character of kind `kind`.

Alternatively, the obsolete form for character declaration

```
character*len  name
```

may be seen in older code, declaring `name` to be of length `len` and default character kind.

Declaration of a variable of intrinsic type may be of the form above, but also may use the `type(...)` form:

```
integer i
real x
double precision y
```

is equivalent to (but greatly preferred over)

```
type(integer) i
type(real) x
type(double precision) y
```

Derived data types

Define a new type, `mytype`:

```
type :: mytype
  integer :: int
  real    :: float
end type mytype
```

Declare a variable of type *mytype*:

```
type(mytype) :: foo
```

The components of a derived type can be accessed with the `%` operator[1]:

```
foo%int = 4
foo%float = 3.142
```

A Fortran 2003 feature (not yet implemented by all compilers) allows to define parameterized data types:

```
type, public :: matrix(rows, cols, k)
  integer, len :: rows, cols
  integer, kind :: k = kind(0.0)
  real(kind = k), dimension(rows, cols) :: values
end type matrix
```

The derived type `matrix` has three type parameters which are listed in parentheses following the type name (they are `rows`, `cols`, and `k`). In the declaration of each type parameter it must be

indicated whether they are kind (`kind`) or length (`len`) type parameters.

Kind type parameters, like those of the intrinsic types, must be constant expressions whereas length type parameters, like the length of an intrinsic character variable, may vary during execution.

Note that parameter `k` has a default value, so it may be provided or omitted when a variable of type `matrix` is declared, as follows

```
type (matrix (55, 65, kind=double)) :: b, c ! default parameter provided
type (matrix (rows=40, cols=50)     :: m    ! default parameter omitted
```

The name of a derived type may not be `doubleprecision` or the same as any of the intrinsic types.

---

1. Many people wonder why Fortran uses `%` as the component-access operator, instead of the more common `.`. This is because `.` is already taken by the operator syntax, i.e. `.not.`, `.and.`, `.my_own_operator.`.

Precision of floating point numbers

Floating point numbers of type `real` cannot have any real value. They can represent real numbers up to certain amount of decimal digits.

FORTRAN 77 guaranteed two floating point types and more recent standards guarantee at least two real types. Real variables may be declared as

```
real x
double precision y
```

`x` here is a real of default kind and `y` is a real of kind with greater decimal precision than `x`. In Fortran 2008, the decimal precision of `y` is at least 10 and its decimal exponent range at least 37.

```
 real, parameter             :: single = 1.12345678901234567890
 double precision, parameter :: double = 1.123456789012345678901d0

 print *, single
 print *, double
```

prints

```
   1.12345684
   1.1234567890123457
```

in common compilers using default configuration.

Notice the `d0` in the double precision constant. A real literal containing `d` instead of `e` for denoting the exponent is used to indicate double precision.

```
! Default single precision constant
1.23e45
! Double precision constant
1.23d45
```

Fortran 90 introduced parameterized `real` types using kinds. The kind of a real type is an integer named constant or literal constant:

```
real(kind=real_kind) :: x
```

or just

```
real(real_kind) :: x
```

This statement declares `x` to be of type `real` with a certain precision depending on the value of `real_kind`.

Floating point literals can be declared with a specific kind using a suffix

```
1.23456e78_real_kind
```

The exact value of `real_kind` is not standardized and differs from compiler to compiler. To inquire the kind of any real variable or constant, the function `kind()` can be used:

```
print *, kind(1.0), kind(1.d0)
```

will typically print

```
4 8
```

or

```
1 2
```

depending on the compiler.

Kind numbers can be set in several ways:

1. Single (default) and double precision:

   ```
   integer, parameter :: single_kind = kind(1.)
   integer, parameter :: double_kind = kind(1.d0)
   ```

2. Using the intrinsic function `selected_real_kind([p, r])` to specify required decimal precision. The returned kind has precision of at least `p` digits and allows exponent of at least `r`.

   ```
   integer, parameter :: single_kind = selected_real_kind( p=6, r=37 )
   integer, parameter :: double_kind = selected_real_kind( p=15, r=200 )
   ```

3. Starting with Fortran 2003, pre-defined constants are available through the intrinsic module

`ISO_C_Binding` to ensure that real kinds are inter-operable with the types `float`, `double` or `long_double` of the accompanying C compiler:

```
use ISO_C_Binding

integer, parameter :: single_kind = c_float
integer, parameter :: double_kind = c_double
integer, parameter :: long_kind = c_long_double
```

4. Starting with Fortran 2008, pre-defined constants are available through the intrinsic module `ISO_Fortran_env`. These constants provide real kinds with certain storage size in bits

```
use ISO_Fortran_env

integer, parameter :: single_kind = real32
integer, parameter :: double_kind = real64
integer, parameter :: quadruple_kind = real128
```

If certain kind is not available in the compiler, the value returned by `selected_real_kind()` or the value of the integer constant is `-1`.

Assumed and deferred length type parameters

Variables of character type or of a derived type with length parameter may have the length parameter either *assumed* or *deferred*. The character variable `name`

```
character(len=len) name
```

is of length `len` throughout execution. Conversely the length specifier may be either

```
character(len=*) ...  ! Assumed length
```

or

```
character(len=:) ...  ! Deferred length
```

Assumed length variables assume their length from another entity.

In the function

```
function f(dummy_name)
  character(len=*) dummy_name
end function f
```

the dummy argument `dummy_name` has length that of the actual argument.

The named constant `const_name` in

```
character(len=*), parameter :: const_name = 'Name from which length is assumed'
```

has length given by the constant expression on the right-hand side.

Deferred length type parameters may vary during execution. A variable with deferred length must have either the `allocatable` or `pointer` attribute

```
character(len=:), allocatable :: alloc_name
character(len=:), pointer :: ptr_name
```

Such a variable's length may be set in any of the following ways

```
allocate(character(len=5) :: alloc_name, ptr_name)
alloc_name = 'Name'         ! Using allocation on intrinsic assignment
ptr_name => another_name    ! For given target
```

For derived types with length parameterization the syntax is similar

```
  type t(len)
    integer, len :: len
    integer i(len)
  end type t

  type(t(:)), allocatable :: t1
  type(t(5)) t2

  call sub(t2)
  allocate(type(t(5)) :: t1)

contains

  subroutine sub(t2)
    type(t(*)), intent(out) :: t2
  end subroutine sub

end
```

Literal constants

Program units often make use of literal constants. These cover the obvious cases like

```
print *, "Hello", 1, 1.0
```

Except in one case, each literal constant is a scalar which has type, type parameters and value given by the syntax.

Integer literal constants are of the form

```
1
-1
-1_1   ! For valid kind parameter 1
1_ik   ! For the named constant ik being a valid kind paramter
```

Real literal constants are of the form

```
1.0    ! Default real
1e0    ! Default real using exponent format
1._1   ! Real with kind parameter 1 (if valid)
1.0_sp ! Real with kind paramter named constant sp
1d0    ! Double precision real using exponent format
1e0_dp ! Real with kind named constant dp using exponent format
```

Complex literal constants are of the form

```
(1, 1.)       ! Complex with integer and real components, literal constants
(real, imag)  ! Complex with named constants as components
```

If the real and imaginary components are both integer, the complex literal constant is default complex, and the integer components are converted to default real. If one component is real, the kind parameter of the complex literal constant is that of the real (and the integer component is converted to that real kind). If both components are real the complex literal constant is of kind of the real with the greatest precision.

Logical literal constants are

```
.TRUE.     ! Default kind, with true value
.FALSE.    ! Default kind, with false value
.TRUE._1   ! Of kind 1 (if valid), with true value
.TRUE._lk  ! Of kind named constant lk (if valid), with true value
```

Character literal values differ slightly in concept, in that the kind specifier precedes the value

```
"Hello"       ! Character value of default kind
'Hello'       ! Character value of default kind
ck_"Hello"    ! Character value of kind ck
"'Bye"        ! Default kind character with a '
'''Bye'       ! Default kind character with a '
""            ! A zero-length character of default kind
```

As suggested above, character literal constants must be delimted by apostrophes or quotation marks, and the start and end marker must match. Literal apostrophes can be included by being within quotation mark delimiters or by appearing doubled. The same for quotation marks.

BOZ constants are distinct from the above, in that they specify only a value: they have no type or type parameter. A BOZ constant is a bit pattern and is specified as

```
B'00000'    ! A binary bit pattern
B"01010001" ! A binary bit pattern
O'012517'   ! An octal bit pattern
O"1267671"  ! An octal bit pattern
Z'0A4F'     ! A hexadecimal bit pattern
Z"FFFFFF"   ! A hexadecimal bit pattern
```

BOZ literal constants are limited in where they may appear: as constants in `data` statements and a

selection of intrinsic procedures.

Accessing character substrings

For the character entity

```
character(len=5), parameter :: greeting = "Hello"
```

a substring may be referenced with the syntax

```
greeting(2:4)  ! "ell"
```

To access a single letter it isn't sufficient to write

```
greeting(1)    ! This isn't the letter "H"
```

but

```
greeting(1:1)  ! This is "H"
```

For a character array

```
character(len=5), parameter :: greeting(2) = ["Hello", "Yo!  "]
```

we have substring access like

```
greeting(1)(2:4)  ! "ell"
```

but we cannot reference the non-contiguous characters

```
greeting(:)(2:4)  ! The parent string here is an array
```

We can even access substrings of literal constants

```
"Hello"(2:4)
```

A portion of a character variable may also be defined by using a substring as a variable. For example

```
integer :: i=1
character :: filename = 'file000.txt'

filename(9:11) = 'dat'
write(filename(5:7), '(I3.3)') i
```

Accessing complex components

The complex entity

```
complex, parameter :: x = (1., 4.)
```

has real part `1.` and complex part `4.`. We can access these individual components as

```
real(x)  ! The real component
aimag(x) ! The complex component
x%re     ! The real component
y%im     ! The complex component
```

The `x%.` form is new to Fortran 2008 and not widely supported in compilers. This form, however, may be used to directly set the individual components of a complex variable

```
complex y
y%re = 0.
y%im = 1.
```

Declaration and attributes

Throughout the topics and examples here we'll see many declarations of variables, functions and so on.

As well as their name, data objects may have *attributes*. Covered in this topic are declaration statements like

```
integer, parameter :: single_kind = kind(1.)
```

which gives the object `single_kind` the `parameter` attribute (making it a named constant).

There are many other attributes, like

- `target`
- `pointer`
- `optional`
- `save`

Attributes may be specified with so-called *attribute specification statements*

```
integer i    ! i is an integer (of default kind)...
pointer i    ! ... with the POINTER attribute...
optional i   ! ... and the OPTIONAL attribute
```

However, it is generally regarded to be better to avoid using these attribute specification statements. For clarity the attributes may be specified as part of a single declaration

```
integer, pointer, optional :: i
```

This also reduces the temptation to use implicit typing.

In most cases in this Fortran documentation this single declaration statement is preferred.

Read Data Types online: https://riptutorial.com/fortran/topic/939/data-types

Chapter 5: Execution Control

Examples

If construct

The `if` construct (called a block IF statement in FORTRAN 77) is common across many programming languages. It conditionally executes one block of code when a logical expression is evaluated to true.

```
[name:] IF (expr) THEN
    block
[ELSE IF (expr) THEN [name]
    block]
[ELSE [name]
   block]
END IF [name]
```

where,

- **name** - the name of the if construct (optional)
- **expr** - a scalar logical expression enclosed in parentheses
- **block** - a sequence of zero or more statements or constructs

A construct name at the beginning of an `if then` statement must have the same value as the construct name at the `end if` statement, and it should be unique for the current scoping unit.

In `if` statements, (in)equalities and logical expressions evaluating a statement can be used with the following operators:

```
.LT.  which is <   ! less than
.LE.           <=  ! less than or equal
.GT.           >   ! greater than
.GE.           >=  ! greater than or equal
.EQ.           =   ! equal
.NE.           /=  ! not equal
.AND.              ! logical and
.OR.               ! logical or
.NOT.              ! negation
```

Examples:

```
! simplest form of if construct
if (a > b) then
    c =  b / 2
end if
!equivalent example with alternate syntax
if(a.gt.b)then
   c=b/2
endif
```

```
! named if construct
circle: if (r >= 0) then
    l = 2 * pi * r
end if circle

! complex example with nested if construct
block: if (a < e) then
    if (abs(c - e) <= d) then
        a = a * c
    else
        a = a * d
    end if
else
    a = a * e
end if block
```

A historical usage of the `if` construct is in what is called an "arithmetic if" statement. Since this can be replaced by more modern constructs, however, it is not covered here. More details can be found here.

SELECT CASE construct

A `select case` construct conditionally executes one block of constructs or statements depending on the value of a scalar expression in a `select case` statement. This control construct can be considered as a replacement for computed `goto`.

```
[name:] SELECT CASE (expr)
[CASE (case-value [, case-value] ...) [name]
   block]...
[CASE DEFAULT [name]
   block]
END SELECT [name]
```

where,

- **name** - the name of the `select case` construct (optional)
- **expr** - a scalar expression of type integer, logical, or character (enclosed in parentheses)
- **case-value** - one or more scalar integer, logical, or character initialization expressions enclosed in parentheses
- **block** - a sequence of zero or more statements or constructs

Examples:

```
! simplest form of select case construct
select case(i)
case(:-1)
    s = -1
case(0)
    s = 0
case(1:)
    s = 1
case default
    print "Something strange is happened"
```

```
end select
```

In this example, `(:-1)` case value is a range of values matches to all values less than zero, `(0)` matches to zeroes, and `(1:)` matches to all values above zero, `default` section involves if other sections did not executed.

Block DO construct

A `do` construct is a looping construct which has a number of iterations governed by a loop control

```
integer i
do i=1, 5
  print *, i
end do
print *, i
```

In the form above, the loop variable `i` passes through the loop 5 times, taking the values 1 to 5 in turn. After the construct has completed the loop variable has the value 6, that is, **the loop variable is incremented once more after the completion of the loop**.

More generally, the `do` loop construct can be understood as follows

```
integer i, first, last, step
do i=first, last, step
end do
```

The loop starts with `i` with the value `first`, incrementing each iteration by `step` until `i` is greater than `last` (or less than `last` if the step size is negative).

It is important to note that since Fortran 95, the loop variable and the loop control expressions must be integer.

An iteration may be ended prematurely with the `cycle` statement

```
do i=1, 5
  if (i==4) cycle
end do
```

and the whole construct may cease execution with the `exit` statement

```
do i=1, 5
  if (i==4) exit
end do
print *, i
```

`do` constructs may be named:

```
do_name: do i=1, 5
end do do_name
```

which is particularly useful when there are nested `do` constructs

```
do1: do i=1, 5
  do j=1,6
    if (j==3) cycle        ! This cycles the j construct
    if (j==4) cycle        ! This cycles the j construct
    if (i+j==7) cycle do1  ! This cycles the i construct
    if (i*j==15) exit do1  ! This exits the i construct
  end do
end do1
```

`do` constructs may also have indeterminate loop control, either "forever" or until a given condition is met

```
integer :: i=0
do
  i=i+1
  if (i==5) exit
end do
```

or

```
integer :: i=0
do while (i<6)
  i=i+1
end do
```

This also allows for an infinite `do` loop via a `.true.` statement

```
print *,'forever'
do while(.true.)
  print *,'and ever'
end do
```

A `do` construct may also leave the order of iterations indeterminate

```
do concurrent (i=1:5)
end do
```

noting that the form of loop control is the same as in a `forall` control.

There are various restrictions on the statements that may be executed within the range of a `do concurrent` construct which are designed to ensure that there are no data dependencies between iterations of the construct. This explicit indication by the programmer may enable greater optimization (including parallelization) by the compiler which may be difficul to determine otherwise.

"Private" variables within an interation can be realized by use of a `block` construct within the `do concurrent`:

```
do concurrent (i=1:5, j=2:7)
  block
    real tempval  ! This is independent across iterations
  end block
end do
```

Another form of the block `do` construct uses a labelled `continue` statement instead of an `end do`:

```
    do 100, i=1, 5
100 continue
```

It is even possible to nest such constructs with a shared termination statement

```
    do 100, i=1,5
    do 100, j=1,5
100 continue
```

Both of these forms, and especially the second (which is obsolescent), are generally to be avoided in the interests of clarity.

Finally, there is also a non-block `do` construct. This is also deemed to be obsolescent and is described elsewhere, along with methods to restructure as a block `do` construct.

WHERE construct

The `where` construct, available in Fortran90 onwards represents a masked `do` construct. The masking statement follows the same rules of the `if` statement, but is applied to all the elements of the given array. Using `where` allows operations to be carried out on an array (or multiple arrays of the same size), the elements of which satisfy a certain rule. This can be used to simplify simultaneous operations on several variables.

Syntax:

```
[name]: where (mask)
    block
[elsewhere (mask)
    block]
[elsewhere
    block]
end where [name]
```

Here,

- **name** - is the name given to the block (if named)
- **mask** - is a logical expression applied to all elements
- **block** - series of commands to be executed

Examples:

```
! Example variables
real:: A(5),B(5),C(5)
A = 0.0
B = 1.0
C = [0.0, 4.0, 5.0, 10.0, 0.0]

! Simple where construct use
where (C/=0)
    A=B/C
elsewhere
    A=0.0
end

! Named where construct
Block: where (C/=0)
    A=B/C
elsewhere
    A=0.0
end where Block
```

Read Execution Control online: https://riptutorial.com/fortran/topic/1657/execution-control

Chapter 6: Explicit and implicit interfaces

Examples

Internal/module subprograms and explicit interfaces

A *subprogram* (which defines a *procedure*), can be either a `subroutine` or a `function`; it is said to be an *internal subprogram* if it is called or invoked from the same `program` or *subprogram* that `contains` it, as follows

```
program my_program

  ! declarations
  ! executable statements,
  ! among which an invocation to
  ! internal procedure(s),
  call my_sub(arg1,arg2,...)
  fx = my_fun(xx1,xx2,...)

contains

  subroutine my_sub(a1,a2,...)
    ! declarations
    ! executable statements
  end subroutine my_sub

  function my_fun(x1,x2,...) result(f)
    ! declarations
    ! executable statements
  end function my_fun

end program my_program
```

In this case the compiler will know all about any internal procedure, since it treats the program unit as a whole. In particular, it will "see" the procedure's `interface`, that is

- whether it is a `function` or `subroutine`,
- which are the names and properties of the arguments `a1`, `a2`, `x1`, `x2`, ...,
- which are the properties of the *result* `f` (in the case of a `function`).

Being the interface known, the compiler can check whether the actual arguments (`arg1`, `arg2`, `xx1`, `xx2`, `fx`, ...) passed to the procedure match with the dummy arguments (`a1`, `a2`, `x1`, `x2`, `f`, ...).

In this case we say that the interface is *explicit*.

A subprogram is said to be *module subprogram* when it is invoked by a statement in the containing module itself,

```
module my_mod

  ! declarations
```

```
contains

  subroutine my_mod_sub(b1,b2,...)
    ! declarations
    ! executable statements
    r = my_mod_fun(b1,b2,...)
  end subroutine my_sub

  function my_mod_fun(y1,y2,...) result(g)
    ! declarations
    ! executable statements
  end function my_fun

end module my_mod
```

or by a statement in another program unit that `use`s that module,

```
program my_prog

  use my_mod

  call my_mod_sub(...)

end program my_prog
```

As in the preceding situation, the compiler will know everything about the subprogram and, therefore, we say that the interface is *explicit*.

External subprograms and implicit interfaces

A subprogram is said to be *external* when it is not contained in the main program, nor in a module or antoher subprogram. In particular it can be defined by means of a programming language other than Fortran.

When an external subprogram is invoked, the compiler cannot access to its code, so all the information allowable to the compiler is implicitly contained in the calling statement of the calling program and in the type an properties of the acutal arguments, not the dummy arguments (whose declaration is unknown to the compiler). In this case we say that the interface is *implicit*.

An `external` statement can be used to specify that a procedure's name is relative to an external procedure,

```
external external_name_list
```

but even so, the interface remain implicit.

An `interface` block can be used to specify the interface of an external procedure,

```
interface
  interface_body
end interface
```

where the `interface_body` is normally an exact copy of the procedure header followed by the declaration of all its arguments and, if it is a function, of the result.

For example, for function `WindSpeed`

```
real function WindSpeed(u, v)
  real, intent(in) :: u, v
  WindSpeed = sqrt(u*u + v*v)
end function WindSpeed
```

You can write the following interface

```
interface
  real function WindSpeed(u, v)
    real, intent(in) :: u, v
  end function WindSpeed
end interface
```

Read Explicit and implicit interfaces online: https://riptutorial.com/fortran/topic/2882/explicit-and-implicit-interfaces

Chapter 7: I/O

Syntax

- `WRITE(unit num, format num)` outputs the data after the brackets in a new line.
- `READ(unit num, format num)` inputs to the variable after the brackets.
- `OPEN(unit num, FILE=file)` opens a file. (There are more options for opening files, but they are not important for I/O.
- `CLOSE(unit num)` closes a file.

Examples

Simple I/O

As an example of writing input & output, we'll take in a real value and return the value and its square until the user enters a negative number.

As specified below, the `read` command takes two arguments: the unit number and the format specifier. In the example below, we use `*` for the unit number (which indicates stdin) and `*` for the format (which indicates the default for reals, in this case). We also specify the format for the `print` statement. One can alternatively use `write(*,"The value...")` or simply ignore formatting and have it as

```
print *,"The entered value was ", x," and its square is ",x*x
```

which will likely result in some oddly spaced strings and values.

```
program SimpleIO
   implicit none
   integer, parameter :: wp = selected_real_kind(15,307)
   real(kind=wp) :: x

   ! we'll loop over until user enters a negative number
   print '("Enter a number >= 0 to see its square. Enter a number < 0 to exit.")'
   do
      ! this reads the input as a double-pricision value
      read(*,*) x
      if (x < 0d0) exit
      ! print the entered value and it's square
      print '("The entered value was ",f12.6,", its square is ",f12.6,".")',x,x*x
   end do
   print '("Thank you!")'

end program SimpleIO
```

Read with some error checking

A modern Fortran example which includes error checking and a function to get a new unit number

for the file.

```
module functions

contains

    function get_new_fileunit() result (f)
        implicit none

        logical    :: op
        integer    :: f

        f = 1
        do
            inquire(f,opened=op)
            if (op .eqv. .false.) exit
            f = f + 1
        enddo

    end function

end module

program file_read
    use functions, only : get_new_fileunit
    implicit none

    integer            :: unitno, ierr, readerr
    logical            :: exists
    real(kind(0.d0))   :: somevalue
    character(len=128) :: filename

    filename = "somefile.txt"

    inquire(file=trim(filename), exist=exists)
    if (exists) then
        unitno = get_new_fileunit()
        open(unitno, file=trim(filename), action="read", iostat=ierr)
        if (ierr .eq. 0) then
            read(unitno, *, iostat=readerr) somevalue
            if (readerr .eq. 0) then
                print*, "Value in file ", trim(filename), " is ", somevalue
            else
                print*, "Error ", readerr, &
                        " attempting to read file ", &
                        trim(filename)
            endif
        else
            print*, "Error ", ierr ," attempting to open file ", trim(filename)
            stop
        endif
    else
        print*, "Error -- cannot find file: ", trim(filename)
        stop
    endif

end program file_read
```

Passing command line arguments

Where command line arguments are supported they can be read in via the `get_command_argument` intrinsic (introduced in the Fortran 2003 standard). The `command_argument_count` intrinsic provides a way to know the number of arguments provided at the command line.

All command-line arguments are read in as strings, so an internal type conversion must be done for numeric data. As an example, this simple code sums the two numbers provided at the command line:

```
PROGRAM cmdlnsum
IMPLICIT NONE
CHARACTER(100) :: num1char
CHARACTER(100) :: num2char
REAL :: num1
REAL :: num2
REAL :: numsum

!First, make sure the right number of inputs have been provided
IF(COMMAND_ARGUMENT_COUNT().NE.2)THEN
  WRITE(*,*)'ERROR, TWO COMMAND-LINE ARGUMENTS REQUIRED, STOPPING'
  STOP
ENDIF

CALL GET_COMMAND_ARGUMENT(1,num1char)   !first, read in the two values
CALL GET_COMMAND_ARGUMENT(2,num2char)

READ(num1char,*)num1                    !then, convert them to REALs
READ(num2char,*)num2

numsum=num1+num2                        !sum numbers
WRITE(*,*)numsum                        !write out value

END PROGRAM
```

The number argument in `get_command_argument` usefully ranges between `0` and the result of `command_argument_count`. If the value is `0` then the command name is supplied (if supported).

Many compilers also offer non-standard intrinsics (such as `getarg`) to access command line arguments. As these are non-standard, the corresponding compiler documentation should be consulted.

Use of `get_command_argument` may be extended beyond the above example with the `length` and `status` arguments. For example, with

```
character(5) arg
integer stat
call get_command_argument(number=1, value=arg, status=stat)
```

the value of `stat` will be `-1` if the first argument exists and has length greater than 5. If there is some other difficulty retrieving the argument the value of `stat` will be some positive number (and `arg` will consist entirely of blanks). Otherwise its value will be `0`.

The `length` argument may be combined with a deferred length character variable, such as in the following example.

```
character(:), allocatable :: arg
integer arglen, stat
call get_command_argument(number=1, length=arglen)  ! Assume for simplicity success
allocate (character(arglen) :: arg)
call get_command_argument(number=1, value=arg, status=stat)
```

Read I/O online: https://riptutorial.com/fortran/topic/6778/i-o

Chapter 8: Intrinsic procedures

Remarks

Many of the available intrinsic procedures have argument types in common. For example:

- a logical argument `MASK` which selects elements of input arrays to be processed
- an integer scalar argument `KIND` which determines the kind of the function result
- an integer argument `DIM` for a reduction function which controls the dimension over which the reduction is performed

Examples

Using PACK to select elements meeting a condition

The intrinsic `pack` function packs an array into a vector, selecting elements based on a given mask. The function has two forms

```
PACK(array, mask)
PACK(array, mask, vector)
```

(that is, the `vector` argument is optional).

In both cases `array` is an array, and `mask` of logical type and conformable with `array` (either a scalar or an array of the same shape).

In the first case the result is rank-1 array of type and type parameters of `array` with the number of elements being the number of true elements in the mask.

```
integer, allocatable :: positive_values(:)
integer :: values(5) = [2, -1, 3, -2, 5]
positive_values = PACK(values, values>0)
```

results in `positive_values` being the array `[2, 3, 5]`.

With the `vector` rank-1 argument present the result is now the size of `vector` (whcih must have at least as many elements as there are true values in `mask`.

The effect with `vector` is to return that array with the initial elements of that array overwritten by the masked elements of `array`. For example

```
integer, allocatable :: positive_values(:)
integer :: values(5) = [2, -1, 3, -2, 5]
positive_values = PACK(values, values>0, [10,20,30,40,50])
```

results in `positive_values` being the array `[2,3,5,40,50]`.

It should be noted that, regardless of the shape of the argument `array` the result is always a rank-1 array.

In addition to selecting the elements of an array meeting a masking condition it is often useful to determine the indices for which the masking condition is met. This common idiom can be expressed as

```
integer, allocatable :: indices(:)
integer i
indices = PACK([(i, i=1,5)], [2, -1, 3, -2, 5]>0)
```

resulting in `indices` being the array `[1,3,5]`.

Read Intrinsic procedures online: https://riptutorial.com/fortran/topic/2643/intrinsic-procedures

Chapter 9: Modern alternatives to historical features

Examples

Implicit variable types

When Fortran was originally developed memory was at a premium. Variables and procedure names could have a maximum of 6 characters, and variables were often *implicitly typed*. This means that the first letter of the variable name determines its type.

- variables beginning with i, j, ..., n are `integer`
- everything else (a, b, ..., h, and o, p, ..., z) are `real`

Programs like the following are acceptable Fortran:

```
program badbadnotgood
  j = 4
  key = 5 ! only the first letter determines the type
  x = 3.142
  print*, "j = ", j, "key = ", key, "x = ", x
end program badbadnotgood
```

You may even define your own implicit rules with the `implicit` statement:

```
! all variables are real by default
implicit real (a-z)
```

or

```
! variables starting with x, y, z are complex
! variables starting with c, s are character with length of 4 bytes
! and all other letters have their default implicit type
implicit complex (x,y,z), character*4 (c,s)
```

Implicit typing is no longer considered best practice. It is very easy to make a mistake using implicit typing, as typos can go unnoticed, e.g.

```
program oops
  real :: somelongandcomplicatedname

  ...

  call expensive_subroutine(somelongandcomplEcatedname)
end program oops
```

This program will happily run and do the wrong thing.

To turn off implicit typing, the `implicit none` statement can be used.

```
program much_better
  implicit none
  integer :: j = 4
  real :: x = 3.142
  print*, "j = ", j, "x = ", x
end program much_better
```

If we had used `implicit none` in the program `oops` above, the compiler would have noticed immediately, and produced an error.

Arithmetic if statement

Arithmetic `if` statement allows one to use three branches depending on the result of an arithmetic expression

```
if (arith_expr) label1, label2, label3
```

This `if` statement transfers control flow to one of the labels in a code. If the result of `arith_expr` is negative `label1` is involved, if the result is zero `label2` is used, and if the result is positive last `label3` is applied. Arithmetic `if` requires all three labels but it allows the re-use of labels, therefore this statement can be simplified to a two branch `if`.

Examples:

```
if (N * N - N / 2) 130, 140, 130

if (X) 100, 110, 120
```

Now this feature is obsolete with the same functionality being offered by the `if` statement and `if-else` construct. For example, the fragment

```
    if (X) 100, 110, 120
100 print*, "Negative"
    goto 200
110 print*, "Zero"
    goto 200
120 print*, "Positive"
200 continue
```

may be written as the `if-else` construct

```
if (X<0) then
  print*, "Negative"
else if (X==0) then
  print*, "Zero"
else
  print*, "Positive"
end if
```

An `if` statement replacement for

```
    if (X) 100, 100, 200
100 print *, "Negative or zero"
200 continue
```

may be

```
if (X<=0) print*, "Negative or zero"
```

Non-block DO constructs

The non-block `do` construct looks like

```
    integer i
    do 100, i=1, 5
100 print *, i
```

That is, where the labelled termination statement is not a `continue` statement. There are various restrictions on the statement that can be used as the termination statement and the whole thing is generally very confusing.

Such a non-block construct can be rewritten in block form as

```
    integer i
    do 100 i=1,5
      print *, i
100 continue
```

or better, using an `end do` termination statement,

```
integer i
do i=1,5
  print *, i
end do
```

Alternate return

Alternate return is a facility to control the flow of execution on return from a subroutine. It is often used as a form of error handling:

```
real x

call sub(x, 1, *100, *200)
print*, "Success:", x
stop

100 print*, "Negative input value"
stop

200 print*, "Input value too large"
```

```
stop

end

subroutine sub(x, i, *, *)
  real, intent(out) :: x
  integer, intent(in) :: i
  if (i<0) return 1
  if (i>10) return 2
  x = i
end subroutine
```

The alternate return is marked by the arguments `*` in the subroutine dummy argument list.

In the `call` statement above `*100` and `*200` refer to the statements labelled `100` and `200` respectively.

In the subroutine itself the `return` statements corresponding to alternate return have a number. This number is not a return value, but denotes the provided label to which execution is passed on return. In this case, `return 1` passes execution to the statement labelled `100` and `return 2` passes execution to the statement labelled `200`. An unadorned `return` statement, or completion of subroutine execution without a `return` statement, passess execution to immediately after the call statement.

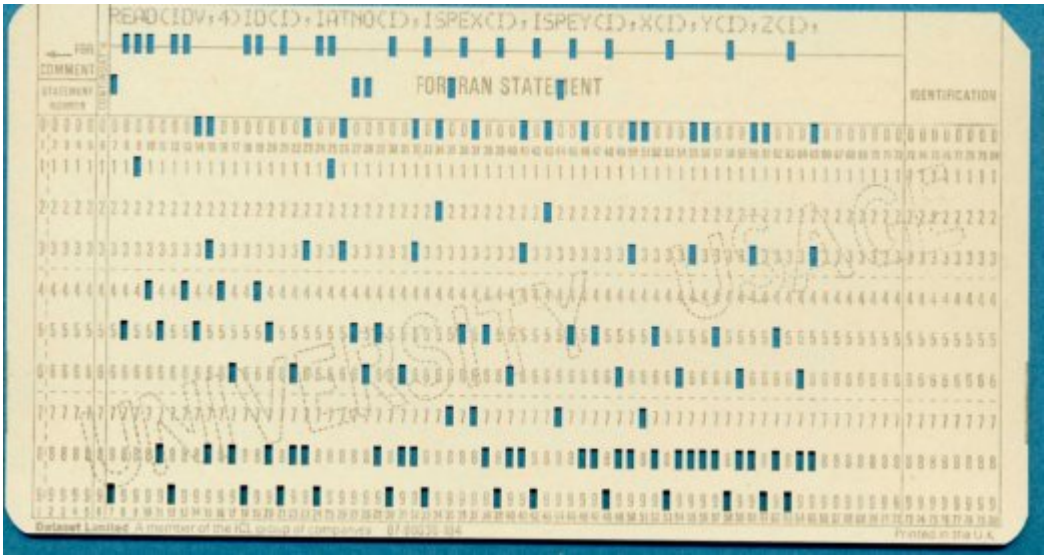
The alternate return syntax is very different from other forms of argument association and the facility introduces flow control contrary to modern tastes. More pleasing flow control can be managed with return of an integer "status" code.

```
real x
integer status

call sub(x, 1, status)
select case (status)
case (0)
  print*, "Success:", x
case (1)
  print*, "Negative input value"
case (2)
  print*, "Input value too large"
end select

end

subroutine sub(x, i, status)
  real, intent(out) :: x
  integer, intent(in) :: i
  integer, intent(out) :: status

  status = 0

  if (i<0) then
    status = 1
  else if (i>10)
    status = 2
  else
    x = i
  end if
```

```
end subroutine
```

Fixed Source Form

Fortran originally was designed for a fixed format form based on an 80 column punched card:



Yes: This is a line of the author's own code

These were created on a card punch machine, much like this:

Images are original photography by the author

The format, as shown on the illustrated sample card, had the first five columns reserved for statement labels. The first column was used to denote comments by a letter **C**. The sixth column was used to denote a statement continuation (by inserting any character other than a zero '0'). The last 8 columns were used for card identification and sequencing, which was pretty valuable if you dropped your deck of cards on the floor! The character coding for punched cards had only a limited set of characters and was upper case only. As a result, Fortran programs looked like this:

```
      DIMENSION A(10)                                                   00000001
C THIS IS A COMMENT STATEMENT TO EXPLAIN THIS EXAMPLE PROGRAM           00000002
      WRITE (6,100)                                                     00000003
 100  FORMAT(169HTHIS IS A RATHER LONG STRING BEING OUTPUT WHICH GOES OVE00000004
```

```
      1R MORE THAN ONE LINE, AND USES THE STATEMENT CONTINUATION MARKER IN00000005
      2COLUMN 6, AND ALSO USES HOLLERITH STRING FORMAT)                    00000006
       STOP                                                                00000007
       END                                                                 00000008
```

The space character was also ignored everywhere, except inside a *Hollerith* character constant (as shown above). This meant that spaces could occur inside reserved words and constants, or completely missed out. This had the side effect of some rather misleading statements such as:

```
       DO 1 I = 1.0
```

is an assignment to the variable `DO1I` whereas:

```
       DO1I = 1,0
```

is actually a `DO` loop on the variable `I`.

Modern Fortran does not now required this fixed form of input and permits free form using any columns. Comments are now indicated by a `!` which can also be appended to a statement line. Spaces are now not permitted anywhere and must be used as separators, much as in most other languages. The above program could be written in modern Fortran as:

```
! This is a comment statement to explain this example program
Print *,"THIS IS A RATHER LONG STRING BEING OUTPUT WHICH no longer GOES OVER MORE THAN ONE
LINE, AND does not need to USE THE STATEMENT CONTINUATION MARKER IN COLUMN 6, or the HOLLERITH
STRING FORMAT"
```

Although the old-style continuation is no longer used, the above example illustrates that very long statements will still occur. Modern Fortran uses a `&` symbol at the end and beginning of the continuation. For example, we could write the above in a more readable form:

```
! This is a comment statement to explain this example program
Print *,"THIS IS A RATHER LONG STRING BEING OUTPUT WHICH still &
         &GOES OVER MORE THAN ONE LINE, AND does need to USE THE STATEMENT &
         &CONTINUATION notation"
```

Common Blocks

In the early forms of Fortran the only mechanism for creating global variable store visible from subroutines and functions is to use the `COMMON` block mechanism. This permitted sequences of variables to be names and shared in common.

In addition to named common blocks there may also be a blank (unnamed) common block.

A blank common block could be declared like

```
common i, j
```

whereas the named block `variables` could be declared like

```
common /variables/ i, j
```

As a complete example, we could imagine a heap store that is used by routines that can add and remove values:

```
      PROGRAM STACKING
      COMMON /HEAP/ ICOUNT, ISTACK(1023)
      ICOUNT = 0
      READ *, IVAL
      CALL PUSH(IVAL)
      CALL POP(IVAL)
      END

      SUBROUTINE PUSH(IVAL)
      COMMON /HEAP/ ICOUNT, ISTACK(1023)
      ICOUNT = ICOUNT + 1
      ISTACK(ICOUNT) = IVAL
      RETURN
      END

      SUBROUTINE POP(IVAL)
      COMMON /HEAP/ ICOUNT, ISTACK(1023)
      IVAL = ISTACK(ICOUNT)
      ICOUNT = ICOUNT - 1
      RETURN
      END
```

Common statements may be used to implicitly declare the type of a variable and to specify the `dimension` attribute. This behaviour alone is often a sufficient source of confusion. Further, the implied storage association and requirements for repeated definitions across program units makes the use of common blocks prone to error.

Finally, common blocks are very restricted in the objects they contain. For example, an array in a common block must be of explicit size; allocatable objects may not occur; derived types must not have default initialization.

In modern Fortran this sharing of variables can be handled by the use of modules. The above example can be written as:

```
module heap
  implicit none
  ! In Fortran 2008 all module variables are implicitly saved
  integer, save :: count = 0
  integer, save :: stack(1023)
end module heap

program stacking
  implicit none
  integer val
  read *, val
  call push(val)
  call pop(val)
```

```
contains
  subroutine push(val)
    use heap, only : count, stack
    integer val
    count = count + 1
    stack(count) = val
  end subroutine push

  subroutine pop(val)
    use heap, only : count, stack
    integer val
    val = stack(count)
    count = count - 1
  end subroutine pop
end program stacking
```

Named and blank common blocks have slightly different behaviours. Of note:

- objects in named common blocks may be defined initially; objects in blank common shall not be
- objects in blank common blocks behave as though the common block has the `save` attribute; objects in named common blocks without the `save` attribute may become undefined when the block is not in the scope of an active program unit

This latter point can be contrasted with the behaviour of module variables in modern code. All module variables in Fortran 2008 are implicitly saved and do not become undefined when the module goes out of scope. Before Fortran 2008 module variables, like variables in named common blocks, would also become undefined when the module went out of scope.

Assigned GOTO

Assigned GOTO uses integer variable to which a statement label is assigned using the ASSIGN statement.

```
100 CONTINUE

...

ASSIGN 100 TO ILABEL

...


GOTO ILABEL
```

Assigned GOTO is obsolescent in Fortran 90 and deleted in Fortran 95 and later. It can be avoided in modern code by using procedures, internal procedures, procedure pointers and other features.

Computed GOTO

Computed GOTO allows branching of the program according to the value of an integer expression.

```
GOTO (label_1, label_2,... label_n) scalar-integer-expression
```

If `scalar-integer-expression` is equal to 1 the program continues at statement label `label_1`, if it is equal to 2 it goes to `label_2` and so on. If it is less then `1` or larger than `n` program continues on next line.

Example:

```
ivar = 2

...

GOTO (10, 20, 30, 40) ivar
```

will jump to statement label 20.

This form of `goto` is obsolescent in Fortran 95 and later, being superseded by the `select case` construct.

Assigned format specifiers

Before Fortran 95 it was possible to use assigned formats for input or output. Consider

```
integer i, fmt
read *, i

assign 100 to fmt
if (i<100000) assign 200 to fmt

print fmt, i

100 format ("This is a big number", I10)
200 format ("This is a small number", I6)

end
```

The `assign` statement assigns a statement label to an integer variable. This integer variable is later used as the format specifier in the `print` statement.

Such format specifier assignment was deleted in Fortran 95. Instead, more modern code can use some other form of execution flow control

```
integer i
read *, i

if (i<100000) then
  print 100, i
else
  print 200, i
```

```
end if

100 format ("This is a big number", I10)
200 format ("This is a small number", I6)

end
```

or a character variable may be used as the format specifier

```
character(29), target :: big_fmt='("This is a big number", I10)'
character(30), target :: small_fmt='("This is a small number", I6)'
character(:), pointer :: fmt

integer i
read *, i

fmt=>big_fmt
if (i<100000) fmt=>small_fmt

print fmt, i

end
```

Statement functions

Consider the program

```
implicit none
integer f, i
f(i)=i

print *, f(1)
end
```

Here `f` is a statement function. It has integer result type, taking one integer dummy argument.[1]

Such a statement function exists within the scope in which it is defined. In particular, it has access to variables and named constants accessible in that scope.

However, statement functions are subject to many restrictions and are potentially confusing (looking at casual glance like an array element assignment statement). Important restrictions are:

- the function result and dummy arguments must be scalar
- the dummy arguments are in the same scope as the function
- statement functions have no local variables
- statement functions cannot be passed as actual arguments

The main benefits of statement functions are repeated by internal functions

```
implicit none

print *, f(1)
```

```
contains

  integer function f(i)
    integer i
    f = i
  end function

end
```

Internal functions are not subject to the restrictions mentioned above, although it is perhaps worth noting that an internal subprogram may not contain further internal subprogram (but it may contain a statement function).

Internal functions have their own scope but also have available host association.

---

[1] In real old code examples, it wouldn't be unusual to see the dummy arguments of a statement function being implicitly typed, even if the result has explicit type.

Read Modern alternatives to historical features online:
https://riptutorial.com/fortran/topic/2103/modern-alternatives-to-historical-features

Chapter 10: Object Oriented Programming

Examples

Derived type definition

Fortran 2003 introduced support for object oriented programming. This feature allows to take advantage of modern programming techniques. Derived types are defined with the following form:

```
TYPE [[, attr-list] :: ] name [(name-list)]
   [def-stmts]
   [PRIVATE statement or SEQUENCE statement]. . .
   [component-definition]. . .
   [procedure-part]
END TYPE [name]
```

where,

- **attr-list** - a list of attribute specifiers
- **name** - the name of derived data type
- **name-list** - a list of type parameter names separated by commas
- **def-stmts** - one or more INTEGER declarations of the type parameters named in the name-list
- **component-definition** - one or more type declaration statements or procedure pointer statements defining the component of derived type
- **procedure-part** - a CONTAINS statement, optionally followed by a PRIVATE statement, and one or more procedure binding statements

Example:

```
type shape
    integer :: color
end type shape
```

Type Procedures

In order to obtain class-like behavior, type and related procedures (subroutine and functions) shall be placed in a module:

Example:

```
module MShape
    implicit none
    private

    type, public :: Shape
    private
        integer :: radius
```

```
    contains
        procedure :: set   => shape_set_radius
        procedure :: print => shape_print
    end type Shape

contains
    subroutine shape_set_radius(this, value)
        class(Shape), intent(in out) :: self
        integer, intent(in)          :: value

        self%radius = value
    end subroutine shape_set_radius

    subroutine shape_print(this)
        class(Shape), intent(in) :: self

        print *, 'Shape: r = ', self%radius
    end subroutine shape_print
end module MShape
```

Later, in a code, we can use this Shape class as follows:

```
! declare a variable of type Shape
type(Shape) :: shape

! call the type-bound subroutine
call shape%set(10)
call shape%print
```

Abstract derived types

An extensible derived type may be *abstract*

```
type, abstract :: base_type
end type
```

Such a derived type may never be instantiated, such as by

```
type(base_type) t1
allocate(type(base_type) :: t2)
```

but a polymorphic object may have this as its declared type

```
class(base_type), allocatable :: t1
```

or

```
function f(t1)
  class(base_type) t1
end function
```

Abstract types may have components and type-bound procedures

```
type, abstract :: base_type
  integer i
contains
  procedure func
  procedure(func_iface), deferred :: def_func
end type
```

The procedure `def_func` is a *deferred* type-bound procedure with interface `func_iface`. Such a deferred type-bound procedure must be implemented by each extending type.

Type extension

A derived type is *extensible* if it has neither the `bind` attribute nor the `sequence` attribute. Such a type may be extended by another type.

```
module mod

  type base_type
    integer i
  end type base_type

  type, extends(base_type) :: higher_type
    integer j
  end type higher_type

end module mod
```

A polymorphic variable with declared type `base_type` is type compatible with type `higher_type` and may have that as dynamic type

```
class(base_type), allocatable :: obj
allocate(obj, source=higher_type(1,2))
```

Type compatability descends through a chain of children, but a type may extend only one other type.

An extending derived type inherits type bound procedures from the parent, but this can be overriden

```
module mod

  type base_type
  contains
    procedure :: sub => sub_base
  end type base_type

  type, extends(base_type) :: higher_type
  contains
    procedure :: sub => sub_higher
  end type higher_type

contains

  subroutine sub_base(this)
```

```
    class(base_type) this
  end subroutine sub_base

  subroutine sub_higher(this)
    class(higher_type) this
  end subroutine sub_higher

end module mod

program prog
  use mod

  class(base_type), allocatable :: obj

  obj = base_type()
  call obj%sub

  obj = higher_type()
  call obj%sub

end program
```

Type constructor

Custom constructors can be made for derived types by using an interface to overload the type name. This way, keyword arguments that don't correspond to components can be used when constructing an object of that type.

```
module ball_mod
  implicit none

  ! only export the derived type, and not any of the
  ! constructors themselves
  private
  public :: ball

  type :: ball_t
     real :: mass
  end type ball_t

  ! Writing an interface overloading 'ball_t' allows us to
  ! overload the type constructor
  interface ball_t
     procedure :: new_ball
  end interface ball_t

contains

  type(ball_t) function new_ball(heavy)
    logical, intent(in) :: heavy

    if (heavy) then
       new_ball%mass = 100
    else
       new_ball%mass = 1
    end if

  end function new_ball
```

```
end module ball_mod

program test
  use ball_mod
  implicit none

  type(ball_t) :: football
  type(ball_t) :: boulder

  ! sets football%mass to 4.5
  football = ball_t(4.5)
  ! calls 'ball_mod::new_ball'
  boulder = ball_t(heavy=.true.)
end program test
```

This can be used to make a neater API than using separate initialisation routines:

```
subroutine make_heavy_ball(ball)
  type(ball_t), intent(inout) :: ball
  ball%mass = 100
end subroutine make_heavy_ball

...

call make_heavy_ball(boulder)
```

Read Object Oriented Programming online: https://riptutorial.com/fortran/topic/2374/object-oriented-programming

Chapter 11: Procedures - Functions and Subroutines

Remarks

Functions and *subroutines*, in conjunction with *modules*, are the tools to break down a *program* into units. This makes the program more readable and manageable. Each one of these units can be thought of as part of the code that, ideally, could be compiled and tested in isolation. The main program(s) can call (or invoke) such subprograms (functions or subroutines) to accomplish a task.

Functions and subroutines are different in the following sense:

- **Functions** return a single object and - usually - don't alter the values of its arguments (i.e. they act just like a mathematical function!);
- **Subroutines** usually perform a more complicated task and they ordinarily alter their arguments (if any is present), as well as other variables (e.g. those declared in the module that contains the subroutine).

Functions and subroutines collectively go under the name of *procedures*. (In the following we will use the verb "call" as synonym of "invoke" even if technically the procedures to be `call`ed are `subroutine`s, whereas `function`s appear as right hand side of assignment or in expressions.)

Examples

Function syntax

Functions can be written using several types of syntax

```
function name()
  integer name
  name = 42
end function
```

```
integer function name()
  name = 42
end function
```

```
function name() result(res)
  integer res
  res = 42
end function
```

Functions return values through a *function result*. Unless the function statement has a `result` clause the function's result has the same name as the function. With `result` the function result is that given by the `result`. In each of the first two examples above the function result is given by `name` ; in the third by `res`.

The function result must be defined during execution of the function.

Functions allow to use some special prefixes.

Pure function means that this function has no side effect:

```
pure real function square(x)
  real, intent(in) :: x
  square = x * x
end function
```

Elemental function is defined as scalar operator but it can be invoked with array as actual argument in which case the function will be applied element-wise. Unless the `impure` prefix (introduced in Fortran 2008) is specified an *elemental* function is also a *pure* function.

```
elemental real function square(x)
  real, intent(in) :: x
  square = x * x
end function
```

Return statement

The `return` statement can be used to exit function and subroutine. Unlike many other programming languages it is not used to set the return value.

```
real function f(x)
  real, intent(in) :: x
  integer :: i

  f = x

  do i = 1, 10

    f = sqrt(f) - 1.0

    if (f < 0) then
      f = -1000.
      return
    end if

  end do
end function
```

This function performs an iterative computation. If the value of `f` becomes negative the function returns value -1000.

Recursive Procedures

In Fortran functions and subroutines need to be explicitly declared as *recursive*, if they are to call themselves again, directly or indirectly. Thus, a recursive implementation of the Fibonacci series could look like this:

```
recursive function fibonacci(term) result(fibo)
  integer, intent(in) :: term
  integer :: fibo

  if (term <= 1) then
    fibo = 1
  else
    fibo = fibonacci(term-1) + fibonacci(term-2)
  end if

end function fibonacci
```

Another example is allowed to calculate factorial:

```
recursive function factorial(n)  result(f)
  integer :: f
  integer, intent(in) :: n

  if(n == 0) then
    f = 1
  else
    f = n * f(n-1)
  end if
end function factorial
```

For a function to directly recursively reference itself its definition must use the `result` suffix. It is not possible for a function to be both `recursive` and `elemental`.

The Intent of Dummy Arguments

The `intent` attribute of a dummy argument in a subroutine or function declares its intended use. The syntax is either one of

```
intent(IN)
intent(OUT)
intent(INOUT)
```

For example, consider this function:

```
real function f(x)
  real, intent(IN) :: x

  f = x*x
end function
```

The `intent(IN)` specifies that the (non-pointer) dummy argument `x` may never be defined or become undefined throughout the function or its initialization. If a pointer dummy argument has the attribute `intent(IN)`, this applies to its association.

`intent(OUT)` for a non-pointer dummy argument means that dummy argument becomes undefined on invocation of the subprogram (except for any components of a derived type with default initialization) and is to be set during execution. The actual argument passed as dummy argument must be definable: passing a named or literal constant, or an expression, is not allowed.

Similarly to before, if a pointer dummy argument is `intent(OUT)` the association status of the pointer becomes undefined. The actual argument here must be a pointer variable.

`intent(INOUT)` specifies that the actual argument is definable and is suitable for both passing in and returning data from the procedure.

Finally, a dummy argument may be without the `intent` attribute. Such a dummy argument has its use limited by the actual argument passed.

For example, consider

```
integer :: i = 0
call sub(i, .TRUE.)
call sub(1, .FALSE.)

end

subroutine sub(i, update)
  integer i
  logical, intent(in) :: update
  if (update) i = i+1
end subroutine
```

The argument `i` can have no `intent` attribute which allows both of the subroutine calls of the main program.

Referencing a procedure

For a function or subroutine to be useful it has to be referenced. A subroutine is referenced in a `call` statement

```
call sub(...)
```

and a function within an expression. Unlike in many other languages, an expression does not form a complete statement, so a function reference is often seen in an assignment statement or used in some other way:

```
x = func(...)
y = 1 + 2*func(...)
```

There are three ways to designate a procedure being referenced:

- as the name of a procedure or procedure pointer
- a procedure component of a derived type object
- a type bound procedure binding name

The first can be seen as

```
procedure(), pointer :: sub_ptr=>sub
call sub()   ! With no argument list the parentheses are optional
call sub_ptr()
```

```
end

subroutine sub()
end subroutine
```

and the final two as

```
module mod
  type t
    procedure(sub), pointer, nopass :: sub_ptr=>sub
  contains
    procedure, nopass :: sub
  end type

contains

  subroutine sub()
  end subroutine

end module

use mod
type(t) x
call x%sub_ptr()   ! Procedure component
call x%sub()       ! Binding name

end
```

For a procedure with dummy arguments the reference requires corresponding *actual* arguments, although optional dummy arguments may be not given.

Consider the subroutine

```
subroutine sub(a, b, c)
  integer a, b
  integer, optional :: c
end subroutine
```

This may be referenced in the following two ways

```
call sub(1, 2, 3)   ! Passing to the optional dummy c
call sub(1, 2)      ! Not passing to the optional dummy c
```

This is so-called *positional* referencing: the actual arguments are associated based on the position in the argument lists. Here, the dummy `a` is associated with `1`, `b` with `2` and `c` (when specified) with `3` .

Alternatively, *keyword* referencing may be used when the procedure has an explicit interface available

```
call sub(a=1, b=2, c=3)
call sub(a=1, b=2)
```

which is the same as the above.

However, with keywords the actual arguments may be offered in any order

```
call sub(b=2, c=3, a=1)
call sub(b=2, a=1)
```

Positional and keyword referencing may both be used

```
call sub(1, c=3, b=2)
```

as long as a keyword is given for every argument following the first appearance of a keyword

```
call sub(b=2, 1, 3)  ! Not valid: all keywords must be specified
```

The value of keyword referencing is particularly pronounced when there are multiple optional dummy arguments, as seen below if in the subroutine definition above `b` were also optional

```
call sub(1, c=3)  ! Optional b is not passed
```

The argument lists for type-bound procedures or component procedure pointers with a passed argument are considered separately.

Read Procedures - Functions and Subroutines online:
https://riptutorial.com/fortran/topic/1106/procedures---functions-and-subroutines

Chapter 12: Program units and file layout

Examples

Fortran programs

A complete Fortran program is made up from a number of distinct program units. Program units are:

- main program
- function or subroutine subprogram
- module or submodule
- block data program unit

The main program and some procedure (function or subroutine) subprograms may be provided by a language other than Fortran. For example a C main program may call a function defined by a Fortran function subprogram, or a Fortran main program may call a procedure defined by C.

These Fortran program units may be given be distinct files or within a single file.

For example, we may see the two files:

prog.f90

```
program main
  use mod
end program main
```

mod.f90

```
module mod
end module mod
```

And the compiler (invoked correctly) will be able to associate the main program with the module.

The single file may contain many program units

everything.f90

```
module mod
end module mod

program prog
  use mod
end program prog

function f()
end function f()
```

In this case, though, it must be noted that the function `f` is still an *external function* as far as the main program and module are concerned. The module will be accessible by the main program, however.

Typing scope rules apply to each individual program unit and not to the file in which they are contained. For example, if we want each scoping unit to have no implicit typing, the above file need be written as

```
module mod
  implicit none
end module mod

program prog
  use mod
  implicit none
end program prog

function f()
  implicit none
  <type> f
end function f
```

Modules and submodules

Modules are documented elsewhere.

Compilers often generate so-called *module files*: usually the file containing

```
module my_module
end module
```

will result in a file named something like `my_module.mod` by the compiler. In such cases, for a module to be accessible by a program unit, that module file must be visible before this latter program unit is processed.

External procedures

An external procedure is one which is defined outside another program unit, or by a means other than Fortran.

The function contained in a file like

```
integer function f()
  implicit none
end function f
```

is an external function.

For external procedures, their existence may be declared by using an interface block (to given an explicit interface)

```
program prog
  implicit none
  interface
    integer function f()
  end interface
end program prog
```

or by a declaration statement to give an implicit interface

```
program prog
  implicit none
  integer, external :: f
end program prog
```

or even

```
program prog
  implicit none
  integer f
  external f
end program prog
```

The `external` attribute is not necessary:

```
program prog
  implicit none
  integer i
  integer f
  i = f()   ! f is now an external function
end program prog
```

Block data program units

Block data program units are program units which provide initial values for objects in common blocks. These are deliberately left undocumented here, and will feature in the documentation of historic Fortran features.

Internal subprograms

A program unit which is not an internal subprogram may contain other program units, called *internal subprograms*.

```
program prog
  implicit none
contains
  function f()
  end function f
  subroutine g()
  end subroutine g
end program
```

Such an internal subprogram has a number of features:

- there is host association between entities in the subprogram and the outer program
- implicit typing rules are inherited (`implicit none` is in effect in `f` above)
- internal subprograms have an explicit interface available in the host

Module subprograms and external subprograms may have internal subprograms, such as

```
module mod
  implicit none
contains
  function f()
  contains
    subroutine s()
    end subroutine s
  end function f
end module mod
```

Source code files

A source code file is a (generally) plain text file which is to processed by the compiler. A source code file may contain up to one main program and any number of modules and external subprograms. For example, a source code file may contain the following

```
module mod1
end module mod1

module mod2
end module mod2

function func1()    ! An external function
end function func1

subroutine sub1()   ! An external subroutine
end subroutine sub1

program prog        ! The main program starts here...
end program prog    ! ... and ends here

function func2()    ! An external function
end function func2
```

We should recall here that, even though the external subprograms are given in the same file as the modules and the main program, the external subprograms are not explicitly known by any other component.

Alternatively, the individual components may be spread across multiple files, and even compiled at different times. Compiler documentation should be read on how to combine multiple files into a single program.

A single source code file may contain either fixed-form or free-form source code: they cannot be mixed, although multiple files being combined at compile-time may have different styles.

To indicate to the compiler the source form there are generally two options:

- choice of filename suffix
- use of compiler flags

The compile-time flag to indicate fixed- or free-form source can be found in the compiler's documentation.

The significant filename suffixes are also to be found in the compiler's documentation, but as a general rule a file named `file.f90` is taken to contain free-form source whereas the file `file.f` is taken to contain fixed-form source.

The use of `.f90` suffix to indicate free-form source (which was introduced in the Fortran 90 standard) often tempts the programmer to use the suffix to indicate the language standard to which the source code conforms. For example, we may see files with `.f03` or `.f08` suffixes. This is generally to be avoided: most Fortran 2003 source is also compliant with Fortran 77, Fortran 90/5 and Fortran 2008. Further, many comilers don't automatically consider such suffixes.

Compilers also often offer a built-in code preprocessor (generally based on cpp). Again, a compile-time flag may be used to indicate that the preprocessor should be run before compilation, but the source code file suffix may also indicate such preprocessing requirement.

For case-sensitive filesystems the file `file.F` is often taken to be a fixed-form source file to be preprocessed and `file.F90` to be a free-form source file to be preprocessed. As before, the compiler's documentation should be consulted for such flags and file suffixes.

Read Program units and file layout online: https://riptutorial.com/fortran/topic/2203/program-units-and-file-layout

Chapter 13: Source file extensions (.f, .f90, .f95, ...) and how they are related to the compiler.

Introduction

Fortran files come under a variety of extensions and each of them have a separate meaning. They specify the Fortran release version, code formatting style and the usage of preprocessor directives similar to C programming language.

Examples

Extensions and Meanings

The following are some of the common extensions used in Fortran source files and the functionalities they can work on.

Lowercase f in the extension

These files do not have the features of preprocessor directives similar to C-programming language. They can be directly compiled to create object files. eg: .f, .for, .f95

Uppercase F in the extension

These files do have the features of preprocessor directives similar to C-programming language. The preprocessors are either defined within the files or using C/C++ like header files or both. These files have to be pre-processed to get the lower case extension files which can be used for compiling. eg: .F, .FOR, .F95

.f, .for, .f77, .ftn

These are used for Fortran files that use **Fixed style format** and thus uses **Fortran 77** release version. Since they are lower case extensions, they cannot have preprocessor directives.

.F, .FOR, .F77, .FTN

These are used for Fortran files that use **Fixed style format** and thus uses **Fortran 77** release version. Since they are upper case extensions, they can have preprocessor directives and thus they have to be preprocessed to get the lower case extension files.

.f90, .f95, .f03, .f08 These are used for Fortran files that use **Free style format** and thus uses later release versions of Fortran. The release versions are in the name.

- f90 - Fortran 90
- f95 - Fortran 95

- f03 - Fortran 2003
- f08 - Fortran 2008

Since they are lower case extensions, they cannot have preprocessor directives.

.F90, .F95, .F03, .F08 These are used for Fortran files that use **Free style format** and thus uses later release versions of Fortran. The release versions are in the name.

- F90 - Fortran 90
- F95 - Fortran 95
- F03 - Fortran 2003
- F08 - Fortran 2008

Since they are upper case extensions, they have preprocessor directives and thus they have to be preprocessed to get the lower case extension files.

Read Source file extensions (.f, .f90, .f95, ...) and how they are related to the compiler. online: https://riptutorial.com/fortran/topic/10265/source-file-extensions---f---f90---f95-------and-how-they-are-related-to-the-compiler-

Chapter 14: Usage of Modules

Examples

Module syntax

Module is a collection of type declarations, data declarations and procedures. The basic syntax is:

```
module module_name
  use other_module_being_used

  ! The use of implicit none here will set it for the scope of the module.
  ! Therefore, it is not required (although considered good practice) to repeat
  ! it in the contained subprograms.
  implicit none

  ! Parameters declaration
  real, parameter, public :: pi = 3.14159
  ! The keyword private limits access to e parameter only for this module
  real, parameter, private :: e = 2.71828

  ! Type declaration
  type my_type
    integer :: my_int_var
  end type

  ! Variable declaration
  integer :: my_integer_variable

! Subroutines and functions belong to the contains section
contains

  subroutine my_subroutine
    !module variables are accessible
    print *, my_integer_variable
  end subroutine

  real function my_func(x)
    real, intent(in) :: x
    my_func = x * x
  end function my_func
end module
```

Using modules from other program units

To access entities declared in a module from another program unit (module, procedure or program), the module must be *used* with the `use` statement.

```
module shared_data
  implicit none

  integer :: iarray(4) = [1, 2, 3, 4]
  real :: rarray(4) = [1., 2., 3., 4.]
end module
```

```
program test

  !use statements most come before implicit none
  use shared_data

  implicit none

  print *, iarray
  print *, rarray
end program
```

The `use` statement supports importing only selected names

```
program test

  !only iarray is accessible
  use shared_data, only: iarray

  implicit none

  print *, iarray

end program
```

Entities can be also accessed under different name by using a *rename-list*:

```
program test

  !only iarray is locally renamed to local_name, rarray is still acessible
  use shared_data, local_name => iarray

  implicit none

  print *, local_name

  print *, rarray

end program
```

Further, renaming can be combined with the `only` option

```
program test
  use shared_data, only : local_name => iarray
end program
```

so that only the module entity `iarray` is accessed, but it has the local name `local_name`.

If selected for importing names mark as *private* you can not import them to your program.

Intrinsic modules

Fortran 2003 introduced intrinsic modules which provide access to special named constants, derived types and module procedures. There are now five standard intrinsic modules:

- `ISO_C_Binding`; supporting C interoperability;
- `ISO_Fortran_env`; detailing the Fortran environment;
- `IEEE_Exceptions`, `IEEE_Arithmetic` and `IEEE_Features`; supporting so-called IEEE arithmetic facility.

These intrinsic modules are part of the Fortran library and accessed like other modules except that the `use` statement may have the intrinsic nature explicitly stated:

```
use, intrinsic :: ISO_C_Binding
```

This ensures that the intrinsic module is used when a user-provided module of the same name is available. Conversely

```
use, non_intrinsic :: ISO_C_Binding
```

ensures that that same user-provided module (which must be accessible) is accessed instead of the intrinsic module. Without the module nature specified as in

```
use ISO_C_Binding
```

an available non-intrinsic module will be preferred over the intrinsic module.

The intrinsic IEEE modules are different from other modules in that their accessibility in a scoping unit may change the behaviour of code there even without reference to any of the entities defined in them.

Access control

Accessibility of symbols declared in a module can be controlled using `private` and `public` attributes and statement.

Syntax of the statement form:

```
!all symbols declared in the module are private by default
private

!all symbols declared in the module are public by default
public

!symbols in the list will be private
private :: name1, name2

!symbols in the list will be public
public :: name3, name4
```

Syntax of the attribute form:

```
integer, parameter, public :: maxn = 1000
```

```
real, parameter, private :: local_constant = 42.24
```

Public symbols can be accessed from program units using the module, but private symbols cannot.

When no specification is used, the default is `public`.

The default access specification using

```
private
```

or

```
public
```

can be changed by specifying different access with *entity-declaration-list*

```
public :: name1, name2
```

or using attributes.

This access control also affects symbols imported from another module:

```
module mod1
  integer :: var1
end module

module mod2
  use mod1, only: var1

  public
end module

program test
  use mod2, only: var1
end program
```

is possible, but

```
module mod1
  integer :: var1
end module

module mod2
  use mod1, only: var1

  public
  private :: var1
end module

program test
  use mod2, only: var1
end program
```

is not possible because `var` is private in `mod2`.

Protected module entities

As well as allowing module entities to have access control (being `public` or `private`) modules entities may also have the `protect` attribute. A public protected entity may be use associated, but the used entity is subject to restrictions on its use.

```
module mod
  integer, public, protected :: i=1
end module

program test
  use mod, only : i
  print *, i   ! We are allowed to get the value of i
  i = 2        ! But we can't change the value
end program test
```

A public protected target is not allowed to be pointed at outside its module

```
module mod
  integer, public, target, protected :: i
end module mod

program test
  use mod, only : i
  integer, pointer :: j
  j => i   ! Not allowed, even though we aren't changing the value of i
end program test
```

For a public protected pointer in a module the restrictions are different. What is protected is the association status of the pointer

```
module mod
  integer, public, target :: j
  integer, public, protected, pointer :: i => j
end module mod

program test
  use mod, only : i
  i = 2   ! We may change the value of the target, just not the association status
end program test
```

As with variable pointers, procedure pointers may also be protected, again preventing change of target association.

Read Usage of Modules online: https://riptutorial.com/fortran/topic/1139/usage-of-modules

Credits

S. No	Chapters	Contributors
1	Getting started with Fortran	Alexander Vogt, Community, Enrico Maria De Angelis, Gilles, haraldkl, High Performance Mark, Ingve, innoSPG, milancurcic, packet0, RamenChef, Serenity, Vladimir F, Yossarian
2	Arrays	Enrico Maria De Angelis, francescalus, G.Clavier, Gilles, Serenity, TTT, Vladimir F, Yossarian
3	C interoperability	Serenity, Yossarian
4	Data Types	Alexander Vogt, Enrico Maria De Angelis, francescalus, Vladimir F, Yossarian
5	Execution Control	Enrico Maria De Angelis, francescalus, haraldkl, ptev, Serenity, syscreat, TTT, Vladimir F
6	Explicit and implicit interfaces	Enrico Maria De Angelis, Serenity, Vladimir F
7	I/O	AL-P, Ed Smith, francescalus, Kyle Kanos, TTT
8	Intrinsic procedures	francescalus
9	Modern alternatives to historical features	Brian Tompsett - , d_1999, Enrico Maria De Angelis, francescalus, Serenity, TTT, Vladimir F, Yossarian
10	Object Oriented Programming	Enrico Maria De Angelis, francescalus, syscreat, Yossarian
11	Procedures - Functions and Subroutines	Alexander Vogt, Enrico Maria De Angelis, francescalus, haraldkl , Serenity, Vladimir F, Yossarian
12	Program units and file layout	agentp, francescalus, haraldkl, trblnc
13	Source file extensions (.f, .f90, .f95, ...) and how they are related to the compiler.	Arun
14	Usage of Modules	Alexander Vogt, Enrico Maria De Angelis, francescalus, Serenity, Vladimir F