무료 전자 책

배우기

Fortran

Free unaffiliated eBook created from
Stack Overflow contributors.

#fortran

1
1: Fortran 2
2
2
Examples 2
2
2
3
4
2: C 5
Examples 5
Fortran C 5
Fortran C 5
3: I / O 7
7
Examples 7
I / O 7
7
8
4: 10
Examples 10
10
10
11
11
12
5: 14
Examples 14
14
15
15

17
18
19
20
20
6: 22
Examples 22
/ 22
23
7: 25
Examples 25
25
25
26
27
28
8: 30
Examples 30
30
30
31
: 33
33
33
33
34
34
, 34
34
35
35
35

36
36
36
36
37
: 37
37
38
38
9: 39
39
Examples 39
39
10: (.f, .f90, .f95, ...) . 40
40
Examples 40
40
11: 41
Examples 41
41
SELECT CASE 42
DO 42
WHERE 44
12: 46
Examples 46
46
if 47
DO 47
48
49
50
GOTO 52

GOTO 52
..... 53
..... 53
13: - 55
..... 55
Examples 55
..... 55
return 56
..... 56
..... 56
..... 57
14: 60
Examples 60
..... 60
..... 61
..... 61
..... 62
..... 62
..... 62
..... 64

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

1: Fortran

Fortran . .

, Fortran GNU, Intel, PGI Cray .

66	ASA (ANSI)	1966-03-07
77	,	1978-04-15
90	, ISO ,	1991-06-15
95		1997-06-15
2003		2004-04-04
2008		2010-09-10

Examples

Fortran . . .

Fortran GNU Fortran gfortran. GNU GCC GNU . https://gcc.gnu.org/wiki/GFortranBinaries . Linux gfortran .

.

- PathScale EKopath
- LLVM (DragonEgg)
- Oracle Developer Studio

- Absoft Fortran
-
- NAG
- PGI

HPC IBM Cray .

Fortran 95 . Fortran 2003 Fortran 2008 ACM Fortran Fortran Wiki .

.

`end` . Fortran .

```
end
```

"," .

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

`write` :

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

`program` `program` `.end` `.,` `implicit none` . Fortran :

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

hello world . . . . `fortran` . (, ++, vi, vim, emacs, gedit, kate) `hello.f90` hello . `hello.f90` . (?) .

```
>gfortran -o hello hello.f90
```

hello world . . .

```
>./hello
```

.

```
> Hello world!
```

. "Hello World" .

Fortran . 2 :

```
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
```

. , . C ++ / .

a A . .

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

.

Fortran : https://riptutorial.com/ko/fortran/topic/904/fortran-

2: C

Examples

Fortran C

Fortran 2003 C Fortran (C) . `iso_c_binding` .

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

`intrinsic` .

`iso_c_binding` .

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

C C Fortran .

C char Fortran .

Fortran C . C . `bind` C . Fortran .

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
```

Fortran `howManyGeese()` C (, *?*) `how_many_geese()` Fortran .

Fortran C

`bind` .

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
```

`goose_c` `goose_f` `.Goose` C `type(goose_t)` .

C : https://riptutorial.com/ko/fortran/topic/2184/c--

3: I / O

- `WRITE(unit num, format num)` .
- `READ(unit num, format num)` .
- `OPEN(unit num, FILE=file)` .(I / O .
- `CLOSE(unit num)` .

Examples

I / O

, .

, `read` . `*` ( ) `*` (,). `print` . `write(*,"The value....")`

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

.

```
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
```

Fortran .

```
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
```

`get_command_argument` (Fortran 2003 `get_command_argument` . `command_argument_count` .

. , .

```
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
```

`get_command_argument` number `0` `command_argument_count` . `0` ( ).

(: `getarg`) . .

`get_command_argument` `length` `status` . ,

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

5 `stat` `-1` . `stat` (`arg`). `0` .

`length` .

```
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)
```

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

4:

Examples

Fortran 2003 . . .

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

,

- **attr-list** -
- **name** -
- **name-list** -
- **def-stmts** - name-list INTEGER
- **component-definition** -
- **procedure-part** - CONTAINS PRIVATE

:

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

( ) :

:

```
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
```

Shape .

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

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

.

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

.

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

.

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

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

- .

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

def_func func_iface . - .

bind sequence . .

```
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
```

base_type higher_type

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

.

.

```
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
```

. , .

```
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
```

API .

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

...

call make_heavy_ball(boulder)
```

: https://riptutorial.com/ko/fortran/topic/2374/--

5:

Examples

Fortran .

```
integer
real
character
complex
logical
```

integer , real complex .

character .

logical .true. .false. .

.

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

```
integer(specific_kind)
```

specific_kind .

kind length .

```
character char
```

char 1 ,

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

name len . .

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

name kind kind length len . char 1 kind .

```
character*len  name
```

name len .

---

intrinsic type(...) .

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

( )

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

`mytype` .

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

mytype .

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

`%` [1] .

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

Fortran 2003 ( ) .

```
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
```

`matrix` (`rows` , `cols` `k`). (`kind`) (`len`) .

.

`k` `matrix`

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

`doubleprecision` .

---

1. Fortran `%` . . . `.not.` , `.and.` , `.my_own_operator.` .

`real` . 10 .

FORTRAN 77 2 . .

```
real x
double precision y
```

x y x . Fortran 2008 y 10 37 .

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

 print *, single
 print *, double
```

```
   1.12345684
   1.1234567890123457
```

.

double precision d0 . e d .

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

Fortran 90 real . .

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

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

x real_kind real .

.

```
1.23456e78_real_kind
```

real_kind . kind() .

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

.

4 8

1 2

.

.

1. () :

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

2. `selected_real_kind([p, r])` . `p` `r` .

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

3. Fortran 2003 `ISO_C_Binding` C `float` , `double` `long_double` .

```
use ISO_C_Binding

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

4. Fortran 2008 `ISO_Fortran_env` . .

```
use ISO_Fortran_env

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

`selected_real_kind()` `-1` .

. `name`

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

`len` .

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

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

.

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

dummy `dummy_name` `dummy_name` .

`const_name` in

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

.

. allocatable pointer .

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

.

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

.

```
  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
```

. .

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

, , .

.

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

.

```
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
```

.

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

. . .

```
.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
```

.

```
"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
```

. . . .

BOZ . . BOZ .

```
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 `data` .

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

.

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

.

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

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

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

.

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

.

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

.

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

.

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

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

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

1. 4.. .

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

x%.. Fortran 2008 .

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

, .

. .

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

single_kind parameter ( ).

.

- target

- pointer
- optional
- save

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

.

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

.

.

: https://riptutorial.com/ko/fortran/topic/939/-

6:

Examples

/

( ) `subroutine` `function` . `contains` `program`

```
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
```

. , `interface` "".

- `function` `subroutine` ,
- `a1` , `a2` , `x1` , `x2` , ...
- `f` (`function`) .

(`arg1` , `arg2` , `xx1` , `xx2` , `fx` , ...) (`a1` , `a2` , `x1` , `x2` , `f` , ..) .).

.

,

```
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
```

use ,

```
program my_prog

  use my_mod

  call my_mod_sub(...)

end program my_prog
```

.

. Fortran .

acutal .). .

external .

```
external external_name_list
```

.

interface .

```
interface
  interface_body
end interface
```

interface_body , .

, WindSpeed

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

.

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

: https://riptutorial.com/ko/fortran/topic/2882/-----

7:

Examples

, . .

```
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
```

(,) `use` .

```
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
```

use .

```
program test

  !only iarray is accessible
  use shared_data, only: iarray

  implicit none

  print *, iarray

end program
```

.

```
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
```

only .

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

iarray local_name .

.

Fortran 2003 , . 5 .

- ISO_C_Binding ; C .
- ISO_Fortran_env ; Fortran .
- IEEE_Exceptions , IEEE_Arithmetic IEEE_Features ; IEEE .

Fortran use .

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

.

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

.

```
use ISO_C_Binding
```

.

IEEE .

`private` `public` .

:

```
!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
```

:

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

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

.

`public` .

```
private
```

```
public
```

entity-declaration-list .

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

.

.

```
module mod1
  integer :: var1
end module

module mod2
  use mod1, only: var1

  public
end module

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

```
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
```

`mod2` `var` private .

(`public` `private`) `protect` . .

```
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
```

```
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
```

. .

```
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
```

.

: https://riptutorial.com/ko/fortran/topic/1139/-

8:

Examples

`dimension` () :

```
! 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)
```

/

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

( ) Fortran 2008 15 7.

Fortran . , `bar` .

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

0 C **1** . Fortran .

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

shape `(6, 3)` , `geese(7, -3)` .

2 ( ) `ubound` `lbound` . `lbound(geese,2)` `-3` `ubound(geese,1)` `12` .

`size` . `size(geese, dim = 1)` 6 .

:

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

.

```
! 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
```

.

```
deallocate(foo)
```

allocate .stat .

```
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
```

deallocate stat .

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

status 0 .

rank-1 .

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

[...] Fortran 2003 . .

, .

.

```
[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
```

. . .

```
[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
```

.

```
[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
```

.

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

constant .

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

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

0 .

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

. 0 :

```
[integer :: ]
```

---

1 . . rank-1 `reshape` .

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

.

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

~ .

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

. .

`spread` .

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

.

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

:

`dimension` . Fortran 2008 5 . [1]

-
-
-
-
-

3 1 [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
```

.

. block . , 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
```

`allocatable pointer` . .

```
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
```

.

.

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

.

allocatable pointer . .

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

.

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

.

---

[1] Fortran 2008 . .

[2] .

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

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

,

.

```
real x(10)
```

3 :

1. x ;
2. x(1) ;
3. x(2:6) .

x .print *, SUM(x) ,print *, SIZE(x) x=1 .

(: x).

```
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
```

( ).

.

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

.

( ) .

```
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)
```

. .

., `x(1:1,1)` `x(1:1,1)` `x(1:1,1:1)` 2 .

.,

```
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
```

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

: `x(:)` , x , .

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

x .

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

type(t) x(2)
```

, .

x .

```
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
```

1 . , .

```
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
```

, Fortran .

. ( ) .

```
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
```

:

```
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)
```

, ( ).

```
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).
```

.* / .

```
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
```

( ).

. , .

```
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
```

. . , .

```
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
```

:

. .

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

. .

`[bound1]:[bound2][:stride]` .

```
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
```

(0) , v x . () ( ) (). .

x Y x Y . x Y x Y .

x(10:1:-1) x(1:10:1) x .

-1 . .

```
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)
```

.

- .
- .
- .

, .:

```
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
```

```
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)
```

: https://riptutorial.com/ko/fortran/topic/996/

9:

. :

- MASK
- KIND KIND
- DIM

Examples

intrinsic pack pack . .

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

(, vector).

array mask array ( ) .

, 1 array .

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

positive_values [2, 3, 5] .

vector rank-1 vector (mask true).

vector array .

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

positive_values [2,3,5,40,50] .

array 1 .

. .

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

indices [1,3,5] .

: https://riptutorial.com/ko/fortran/topic/2643/-

10: (.f, .f90, .f95, ...) .

Fortran . Fortran , C .

Examples

Fortran .

f

C . . : .f, .for, .f95

F

C . C / C ++ . . : .F, .FOR, .F95

.f, .for, .f77, .ftn

Fortran **Fortran 77** . .

.F, .FOR, .F77, .FTN

Fortran **Fortran 77** . .

.f90, .f95, .f03, .f08 Free Fortran Fortran . .

- f90 - 90
- f95 - 95
- f03 - Fortran 2003
- f08 - Fortran 2008

.

.F90, .F95, .F03, .F08 Free Fortran Fortran . .

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

.

(.f, .f90, .f95, ...) . : https://riptutorial.com/ko/fortran/topic/10265/-----f---f90---f95---------

11:

Examples

`if` (FORTRAN 77 IF) . .

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

,

- **name** - if ()
- **expr** -
- - 0

`if then . end if . .

`if` , () .

```
.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
```

:

```
! 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
```

`if` " if" . , . .

SELECT CASE

`select case` `select case` . `goto` .

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

,

- **name** - `select case` ()
- **expr** - , ( )
- **case-value** - ,
- - 0

:

```
! 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
```

`(:-1)` 0 `(0)` `(0)` `(1:)` 0 , `default` . .

DO

`do`

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

```
print *, i
```

i 5 15 . 6. .

do .

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

i first step i last (last).

Fortran 95 .

cycle .

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

exit .

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

---

do .

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

do .

```
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 ""

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

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

do .true.

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

---

do

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

forall .

do concurrent . ( ) , .

interaction "private" do concurrent block .

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

---

do end do continue .

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

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

, () .

, non-block do . , , do .

WHERE

Fortran90 where do . masking if . where ( ) , . .

:

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

,

- **name** - ( ).
- - .
- -

:

```
! 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
```

: https://riptutorial.com/ko/fortran/topic/1657/-

12:

Examples

Fortran . 6 . , .

- i, j, ..., n `integer`
- (a, b, ..., h, o, p, ..., z) `real`

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
```

`implicit` `implicit` .

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

```
! 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)
```

. .

```
program oops
  real :: somelongandcomplicatedname

  ...

  call expensive_subroutine(somelongandcomplEcatedname)
end program oops
```

.

`implicit none` .

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

implicit none oops , , .

if

if .

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

if . arith_expr label1 0 label2 label3 . if , if .

:

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

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

if if-else . ,

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

if-else .

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

if

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

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

DO

non-block do .

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

, continue . .

.

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

, end do ,

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

. :

```
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
```

* .

*100 *200 call 100 200 .

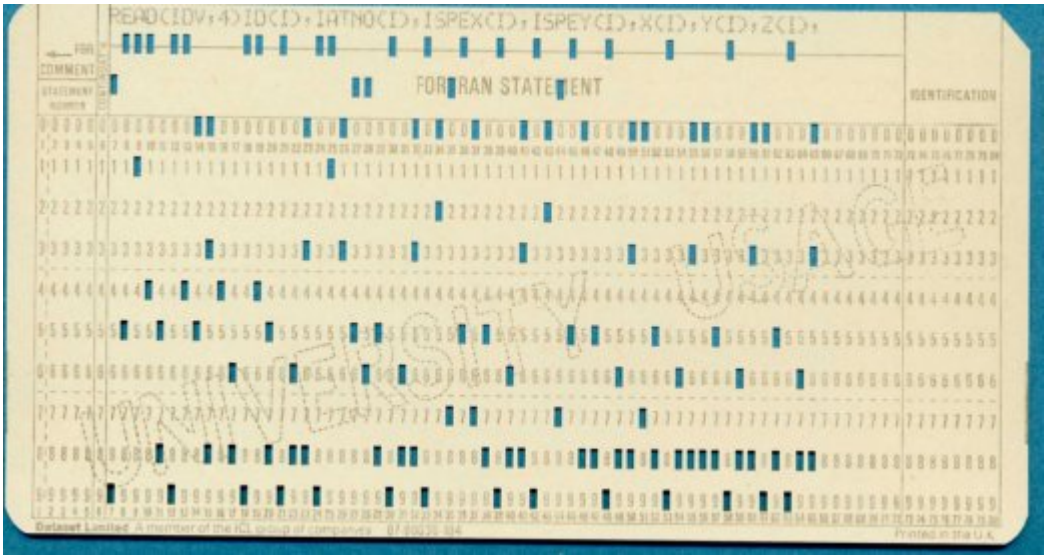
return return . . return 1 100 return 2 200 . return , return , passess .

. "" .

```
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
```

Fortran 80 .



: .

.

.

5 . **C** . ('0'). 8 , . . Fortran .

```
       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
```

Hollerith ( ). . :

```
       DO 1 I = 1.0
```

`DO1I` .

```
       DO1I = 1,0
```

`I DO` .

Modern Fortran . `!` `!` . . Fortran .

```
! 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"
```

. Fortran `&` . .

```
! 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"
```

Fortran COMMON . .

() .

.

```
common i, j
```

variables .

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

.

```
       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
```

---

dimension . ., .

. , . . .

Fortran . .

```
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
```

. :

- .
- `save` . `save`

. Fortran 2008 . Fortran 2008 .

GOTO

GOTO ASSIGN .

```
100 CONTINUE

...

ASSIGN 100 TO ILABEL

...


GOTO ILABEL
```

GOTO Fortran 90 Fortran 95 . , , .

GOTO

GOTO .

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

`scalar-integer-expression` 1 `label_1` 2 `label_2` . `1` `n` .

:

```
ivar = 2
```

```
...

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

20 .

`goto` Fortran 95 , `select case` .

Fortran 95 .

```
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
```

`assign` . `print` .

Fortran 95 .

```
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
```

```
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
```

.

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

print *, f(1)
end
```

f . . [1]

. , .

( ). .

- .
- .
- .
- .

.

```
implicit none

print *, f(1)

contains

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

end
```

( ) , .

.

---

1 .

: https://riptutorial.com/ko/fortran/topic/2103/----

13: -

. . , . () ( ) ( ) .

.

- (, !).
- ( ) (:) .

. ("" " `call subroutine` , `function`).

Examples

.

```
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
```

. `result` . `result` `result` . `name` . `res` .

.

.

.

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

. `impure` (Fortran 2008) .

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

return

return . .

```
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
```

. f -1000 .

Fortran . , :

```
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
```

.

```
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
```

result . recursive elemental .

intent . .

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

, .

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

  f = x*x
end function
```

intent(IN) () x . intent(IN) , .

intent(OUT) ( ) . . .

intent(OUT) . .

intent(INOUT) .

, intent . .

,

```
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
```

i intent .

. call .

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

. .

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

.

-
-
-

.

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

subroutine sub()
end subroutine
```

```
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
```

, .

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

.

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

. . a 1 b 2 c () 3 .

,

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

.

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

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

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

,

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

. b

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

.

- : https://riptutorial.com/ko/fortran/topic/1106/------

14:

Examples

. :

-
-
-
-

( ) Fortran . , C Fortran , Fortran C .

.

.

prog.f90

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

mod.f90

```
module mod
end module mod
```

( ) .

.

.f90

```
module mod
end module mod

program prog
  use mod
end program prog

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

f . .

. .

```
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
```

.

.

```
module my_module
end module
```

`my_module.mod` . , .

Fortran .

.

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

.

( )

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

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

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

`external` .

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

. , .

.

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

.

- .
- (implicit none f)
- .

.

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

() . . , .

```
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
```

.

. .

. .

.

-
-

.

, `file.f90` `file.f90` `file.f` `file.f` .

(Fortran 90) `.f90` . , `.f03` `.f08` . Fortran 2003 Fortran 77, Fortran 90/5 Fortran 2008 .
.

(cpp) . .

/ `file.F` `file.F90` . , .

: https://riptutorial.com/ko/fortran/topic/2203/----

S. No		Contributors
1	Fortran	Alexander Vogt, Community, Enrico Maria De Angelis, Gilles, haraldkl, High Performance Mark, Ingve, innoSPG, milancurcic, packet0, RamenChef, Serenity, Vladimir F, Yossarian
2	C	Serenity, Yossarian
3	I / O	AL-P, Ed Smith, francescalus, Kyle Kanos, TTT
4		Enrico Maria De Angelis, francescalus, syscreat, Yossarian
5		Alexander Vogt, Enrico Maria De Angelis, francescalus, Vladimir F, Yossarian
6		Enrico Maria De Angelis, Serenity, Vladimir F
7		Alexander Vogt, Enrico Maria De Angelis, francescalus, Serenity, Vladimir F
8		Enrico Maria De Angelis, francescalus, G.Clavier, Gilles, Serenity, TTT, Vladimir F, Yossarian
9		francescalus
10	(.f, .f90, .f95, ...) .	Arun
11		Enrico Maria De Angelis, francescalus, haraldkl, ptev, Serenity, syscreat, TTT, Vladimir F
12		Brian Tompsett - , d_1999, Enrico Maria De Angelis, francescalus, Serenity, TTT, Vladimir F, Yossarian
13	-	Alexander Vogt, Enrico Maria De Angelis, francescalus, haraldkl , Serenity, Vladimir F, Yossarian
14		agentp, francescalus, haraldkl, trblnc