Fortran 90/95 reference

As I started using Fortran, I found a number of references online, but none were completely satisfactory to me. The best ones I've found are

- Introduction to Fortran 90 at Queen's University of Belfast
- Fortran 90 for the Fortran 77 Programmer

The textbook *Fortran 90 Programming*, by Ellis, Philips, Lahey was also valuable.

Being familiar with C/C++, though, there was little comparison with features in C. For instance, are the logical and/or operators short-circuiting, as they are in C? (Not really.) Is "if (1)" valid? (It is not.) I also found many of the online examples to be poorly formatted and hard to read. Minimally, code should be lowercase, not shouting in uppercase, and indented. Some documents listed things like intrinsic functions with little or no explanation. I really needed an online quick reference to check on legal and recommended syntax. I also find that writing a guide is a great way to learn all the subtleties of a language. Hence this guide.

I do not claim that everything in this guide is 100% correct. It is certainly not complete. Particularly, I will describe only a subset of Fortran 90, omitting most archaic features of Fortran 77.

Contents

Conventions

Syntax highlighting styles

keywords
keyword operators
comments
`strings`
types
`[optional stuff]` (in brackets)
expression to fill in (in italics)
`...statements to fill in...` (ellipsis)

In this guide I indent by 4 spaces, though in real code I indent by a tab, and set my editor to display tabs as 4 spaces. I tend to use spaces according to these guidelines:

- to line up similar pieces of code
- after commas, following English usage
- *not* after commas in simple indices, like `A(i,j)`
- around binary operators +, –, <, >, =, etc.
- *not* around binary operators **, *, / which have the highest precedence
- *not* around = in variable declarations or argument passing
- as necessary to improve readability, especially for complicated code

Basics

The basic structure of a Fortran program is:

```
program name
    implicit none
    ...variable declarations...
    ...statements...
end [program [name]]
```

This is equivalent to C's main function. The name must be unique, but is otherwise unused.

`implicit none` fixes an old, very bad misfeature of FORTRAN 77. It should always be used in every program, interface, and module. Procedures contained in a module inherit implicit none from the module. Procedures declared outside a module, for instance in the same file as program, do not inherit implicit none—a good reason to put everything in modules. Interfaces apparently never inherit implicit none, so should have their own.

Like the older C standard, all variables must be declared at the top of the program or procedure. Procedure arguments are also declared there, similar to old style K&R C.

Fortran is case-*in*sensitive. I set everything in lowercase, because lowercase is easier to read. Occasionally I use uppercase or TitleCase for some variables where that makes sense, such as for matrices.

Identifiers start with a letter, contain letters, digits, and underbars, and have up to 31 characters. That is, they match [a-zA-Z]\w{0,30}

Lines can be 132 characters long. Lines ending with & ampersand are continued on the next line. An optional leading & ampersand on the next line is ignored, except within a string. If a string is continued on the next line, it starts again immediately after the required initial & ampersand. (But I prefer concating strings, rather than a string spanning multiple lines.)

Semicolons separate statements on a single line, like C and Python. (But no terminating semicolon is required, like Python, unlike C.)

Comments start with ! and go to the end of the line, like C's //, Python's #, Matlab's %. There is no explicit comment terminator, unlike C's /* ... */.

Variable declaration

Variables are declared with their type and a comma separated list of variable names. Options may also be included.

```
type:: names
type, options:: names
```

`type` is one of:

`logical`	boolean taking values `.true.` or `.false.`, like C's bool
`integer`	usually 32 or 64 bit, like C's int
`real`	usually 32 bit, like C's float
`double precision`	usually 64 bit, like C's double
`character(len=len)`	strings
`type(name)`	user defined types, like C's structs

For local variables, `options` is comma separated list of:

`parameter`	constants, like C's const
`allocatable`	an array that will be dynamically allocated
`dimension(...)`	declare shape of array
`save`	retain value between calls

For subroutine arguments, `options` may also include `intent` and `optional`

Variables can be initialized when declared, for instance

```
integer:: a=0, b=5
```

but this implicitly adds save, making them static! That is, they are initialized only the first time a function is called, while subsequent calls retain the value from the previous call. (A horrible design choice.) See bugs and Fortran bug bites.

Fortran 90 has no concept of unsigned integers, nor 1 byte or 2 byte integers. It has a single, signed integer type, typically of 4 or 8 bytes. (Fortran 2003 adds new types for C interoperability.)

Arrays

Arrays can have up to 7 dimensions, specified within ( ) parenthesis. The dimensions are specified as part of the variable or using the dimension keyword. For instance,

```
real:: v(4), A(4,6)
real, dimension(50):: x, y
```

declares vector v of length 4, matrix A with 4 rows and 6 columns, vectors x and y of length 50. (Ellis prefers the dimension syntax. I prefer the parenthesis syntax, which closely matches with C syntax.) You can also declare the lower and upper bound (this is impossible in C and most other languages), like

```
real:: x(-50:-1), y(-25:24)
```

If the length is unknown, use : colon in its place. This is used for allocatable arrays, and for *assumed-shape array* arguments in procedures.

```
real:: arg(:)
real, allocatable:: x(:), B(:,:)
```

The older `array(*)` syntax is for *assumed-size arrays*. It is far less powerful than assumed-shape, so should be avoided in new code.

Array initialization

Arrays can be initialized with a list of values, delimited by `(/` and `/)`, such as

```
x = (/ 1, 2, 3.5, 4.2 /)
```

In Fortran 2003, you can use square brackets, `x = [ 1, 2, 3 ]`

For multi-dimensional matrices, use reshape to get the correct dimensions. Values are taken *column-wise*, so each row below becomes a column:

```
A = reshape( (/ &
  1, 2, 3,      &
  4, 5, 6       &
```

```
    /), (/ 3, 2 /)
```

initializes the matrix

```
    A = [
      1,  4
      2,  5
      3,  6
    ]
```

The *implied do* syntax creates a list of values. Implied do loops can be nested. (Note the i and j are just dummy loop counters, and *not* the array indices.) An implied do loop can be one element in a list.

	Implied do notation	Equivalent to

Simple	`x = (/ (2*i, i=1,4) /)`	**do** i = 1,4
		    x(i) = 2*i
		end do

Nested	`x = (/ ((i*j, i=1,4), j=1,6) /)`	k = 1
		do j = 1,6
		    do i = 1,4
		        x(k) = i*j
		        k = k+1
		    end do
		end do

Item in list	`x = (/ 0, 0, (i, i=3,5), 0, 0 /)`	`x = (/ 0, 0, 3, 4, 5, 0, 0 /)`

Array operations

Arrays with the same shape (i.e. same number of dimensions, and same length in each dimension) are called *conformable*. A scalar is conformable with any array. Many operators and intrinsic functions may be applied element-wise to conformable arrays. Some examples:

	Matrix notation	Equivalent to

Element-wise multiplication	`z = x*y`	**do** i = 1,4
		    z(i) = x(i)*y(i)
		end do

Initialization	`A = 0.`	**do** i = 1,4
		    do j = 1,6
		        A(i,j) = 0.
		    end do
		end do

Element-wise function	`z = sin(x)`	**do** i = 1,4
		    z(i) = sin(x(i))
		end do

Element-wise function	`z = max( 100., x, y )`	**do** i = 1,4
		    z(i) = max(100., x(i), y(i))
		end do

Strings

Note strings are fixed length. Text that is shorter is padded on right with spaces, while text that is longer is truncated.

Strings are enclosed in `'...'` single quotes or `"..."` double quotes.
To include an ' apostrophe in a string, use double quotes like `"foo'bar"`, or two `''` apostrophes like `'foo''bar'`.
To include a " quote in a string, use single quotes like `'foo"bar'`, or two `""` quotes like `"foo""bar"`.

Example of passing array of strings:

```
    character(len=*), dimension(:):: strings
    character(len=len(strings)):: str
```

You frequently have to trim strings to remove padding when printing them. For instance,

```
    print '(a,a,a)', "Error opening file '", trim(filename), "'"
```

Derived Types

To create a data structure, use type, which is similar to C's struct. Types should be declared in modules, in order to be used by multiple subroutines. Types may be nested. A derived type can be initialized with a structure constructor listing all the components, very similar to initializing a struct in C.

Individual members are accessed with a % percent, as they would be with a . period in C. Example:

```
    type person
        character(len=20):: first, last
        integer::           birthyear
        character(len=1)::  gender
    end type

    type employee
        type(person)::      person
        integer::           hire_date
        character(len=20):: department
    end type

    type(person):: jack
    type(employee):: jill
    jack = person( "Jack", "Smith", 1984, "M" )
    jill = employee( person( "Jill", "Smith", 1984, "F" ), 2003, "sales" )

    print *, jack%first, jack%last
    print *, jill%person%first, jill%person%last, jill%department
```

Operators

From highest to lowest precedence. Groups of equal precedence are shown by color and dividing lines; within a group, precedence is left-to-right. Parenthesis obviously supersede order of operations.

Operator		Description	Example
`**`		exponent	`a**b`
`*`		multiply	`a*b`
`/`		divide	`a/b`
`+`		add	`a + b`
`-`		subtract	`a - b`
`//`		string concatenation	`"foo" // "bar"`
`<`	`.lt.`	less than	`a < b`
`<=`	`.le.`	less than or equals	`a <= b`
`>`	`.gt.`	greater than	`a > b`
`>=`	`.ge.`	greater than or equals	`a >= b`
`==`	`.eq.`	equals	`a == b`
`/=`	`.ne.`	not equals †	`a /= b`
`.not.`		logical not (unary)	`.not. a`
`.and.`		logical and	`a .and. b`
`.or.`		logical or	`a .or. b`
`.eqv.`		logical = equals	`a .eqv. b`
`.neqv.`		xor, logical ≠ not equals	`a .neqv. b`

Logical operators *can be* be short-circuiting, as in C, but apparently are *not required* to be short-circuiting. See the Fortran 77 standard and Fortran bug bites. That is, if expression A is false when evaluating `(A .and. B)`, then expression B *might* not be evaluated. Similarly, if A is true when evaluating `(A .or. B)`, then B *might* not be evaluated.

Coming from C/C++, Perl, Python, this ambiguity is annoying. Particularly it makes some conditional expressions difficult. For instance, in C it is common to check indices before using them:
`if ( i < len && a[i] == 'z' ) { ... }`
but this sometimes fails in Fortran. A nested if-then works, but that may require repeating the else clause. Alternatively, make a function to evaluate particular cases correctly, such as:

```
    ! ----------------------------------------
    ! Return str(i:j) if 1 <= i <= j <= len(str),
    ! else return empty string.
    ! This is needed because Fortran allows but doesn't *require* short-circuited
    ! logical AND and OR operators. So this sometimes fails:
    !     if ( i < len(str) .and. str(i+1:i+1) == ':' ) then
    ! but this works:
    !     if ( substr(str, i+1, i+1) == ':' ) then

    character function substr( str, i, j )
        ! arguments
        character(len=*), intent(in):: str
        integer, intent(in):: i, j

        if ( 1 <= i .and. i <= j .and. j <= len(str)) then
            substr = str(i:j)
        else
            substr = ''
        endif
    end function substr
```

Logical operators operate only on logical expressions, unlike C. For instance, `false < true` is true in C, but is not legal in Fortran. Conversely, arithmetic operators do not operate on logical expressions, hence in C, `true == true`, but in Fortran, `.true. .eqv. .true.` (though you could overload == for logicals).

† Fortran's /= is horrid syntax for C programmers, where /= is divide-by, but it is a nice mnemonic with the slash in ≠.

Conditionals

Fortran has single line if statements:

```
    if ( logical_expr ) statement
```

and block if-else statements:

```
    if ( logical_expr ) then
        ...
    else if ( logical_expr ) then
        ...
    else
        ...
    end if
```

Here, logical_expr must have a logical (true/false) value (I think like Java). Numeric and pointer expressions are not allowed, unlike C, Python, etc.

For multiple mutually-exclusive cases of an expression, there is the select statement, similar to C's switch statement, but more general. Cases do not "fall through," unlike C, but a list of values can be specified for each case.

```
    select case( expr )
        case( values )
            ...
        case default
            ...
    end select
```

`expr` can be an integer, string, or logical expression, but *not* a real expression.
`values` can be a single value, or a range, or a list of values and ranges. To indicate a range, use
`lower:` to specify lower bound of range (i.e. lower ≤ expr),
`:upper` to specify upper bound of range (i.e. expr ≤ upper), or
`lower:upper` to specify both lower and upper bounds (i.e. lower ≤ expr ≤ upper). For example, my schedule might be

```
    select case( hour )
        case( 1:8 )
            activity = 'sleep'
        case( 9:11, 13, 14 )
            activity = 'class'
        case( 12, 17 )
            activity = 'meal'
        case default
            activity = 'copious free time'
    end select
```

Loops

The do loop is similar to C's for loop.

```
    do variable = first, last[, increment]
        ...
    end do
```

The increment may be positive or negative; it defaults to 1. If increment > 0, its C equivalent is

```
    for( variable = first; variable <= last; variable += increment ) { ... }
```

If increment < 0, its C equivalent is

```
    for( variable = first; variable >= last; variable += increment ) { ... }
```

The loop variable may not be modified inside the loop.

Use **`exit`** to immediately leave a loop, like C's break or Perl's last.
Use **`cycle`** to skip the rest of the current iteration, like C's continue or Perl's next.

There is also a do loop with no counter. You must use exit (or return) to leave the loop.

```
    do
        ...
        if ( logical_expr ) exit
        ...
        if ( logical_expr ) cycle
        ...
    end do
```

exit and cycle take optional labels to specify which loop to exit/cycle, so you can exit an outer loop. (This is not possible in C without resorting to goto (gasp!). Perl has similar loop labels.)

```
outer: do
inner:     do
               ...
```

```
                if ( logical_expr ) exit outer
                ...
                if ( logical_expr ) exit  ! applies to inner loop by default
                ...
            end do inner
        end do outer
```

There is a do-while construct, like C's while loop.

```
    do while( logical_expr )
        ...
    end do
```

which is equivalent to

```
    do
        if ( .not. logical_expr ) break
        ...
    end do
```

(Ellis recommends against while loops, and the Queen's University intro omits them, but they exactly match the while loop in C, Python, Perl, etc.)

Procedures

Fortran has 2 types of procedures: functions, which return a value (and usually should not modify their arguments), and subroutines, which presumably modify their arguments or have other side effects. A subroutine is like a void function in C.

Subroutine syntax

```
    subroutine name( args )
        ...argument declarations...
        ...local variable declarations...
        ...statements...
    end [subroutine [name]]
```

`return` exits the function or subroutine immediately, like C's return. However, it does not take the return value, unlike C's return.

Function syntax

```
    type function name( args )
        ...argument declarations...
        ...local variable declarations...
        ...statements...
    end [function [name]]
```

Here I've specified the return type before the function keyword. It can also be declared as if it were an argument variable, but I find this counter-intuitive. To specify the return value, assign to the function's name, as if it were a variable. Unlike C and C++, Fortran can handle returning arrays and composite types with no difficulty. (E.g. in C++, returning a local array causes a dangling pointer, since it gets deallocated when the function exits. Fortran does not have this problem.)

Declaring arguments

Argument variables are declared as usual, but may include an additional intent, which says whether a variable is input, output, or both input & output.

`intent(in)`	variable may not be changed, like C's const
`intent(out)`	variable may not be referenced before being set, no equivalent in C
`intent(inout)`	no constraints

The reason for all the intents is that Fortran passes everything by reference, so it would be very easy to accidentally modify some scalar variable. In C, scalars are often passed by value, so you don't have to worry about a procedure modifying a scalar variable in the calling procedure.

Arguments may also be optional, using the `optional` keyword. To test whether an optional argument was passed in, use the present(*argument*) function.

Optional arguments differ from C++ and Python in that no default value is assigned. They must explicitly be tested for being present or not. If not present, an alternate variable may be needed to store the default value. (This seems cumbersome to me, but is useful when there is no good default value.)

Passing array arguments

To pass arrays into a procedure, use assumed-shape arrays, where a colon : is used in place of the unknown lengths of each dimension. Assumed-shape arrays **absolutely require** the procedure to be in a module, which generates an implicit interface, or to have an explicit interface. Without an interface, it will compile but fail at runtime, since the caller doesn't know to use assumed-shape semantics.

The size function queries for the total number of elements in an array. The lbound and ubound functions query for the lower and upper bounds for a particular dimension.

```
    real:: x(:)
    real:: A(:,:)

    m = ubound(A,1)
    n = ubound(A,2)

    n = size(array)
    n = lbound(array, dimension)
```

```
n = ubound(array, dimension)
```

Returning array arguments

To return an array, the return type, including its size, must be declared along with the other arguments. It cannot be put before the function keyword. For instance:

```
function matrix_multiply( A, B )
    real:: A(:,:), B(:,:)
    real:: matrix_multiply( ubound(A,1), ubound(B,2) )

    matrix_multiply = matmul( A, B )
end function matrix_multiply
```

Modules

Modules group together related procedures and variables. It can also function similarly to a C header file by listing interfaces of external library functions. Variables declared in the module are global to the entire module. Procedures and variables can be declared public or private.

```
module name
    implicit none
    save
    [public | private]
    ...module variable declarations...
    ...interfaces declarations...
contains
    ...procedure definitions...
end [module [name]]
```

If there are no procedures, then `contains` is not required.

Another program or procedure uses the variables and procedures in a module with the use module command. By default, all of the module's public variables and procedures are imported into the current namespace. You can also limit which variables and procedures to import using the only syntax. You can locally rename a module's variable using => syntax, similar to Python's `import bar.b as c`. (This is fine for some purposes, but for resolving name conflicts I find it ugly. See my wishlist regarding namespaces.)

```
module bar
    implicit none
    integer:: a, b

    procedure baz( b )
        integer:: b
        print *, b
    end procedure baz
end module bar

procedure foo()
    use bar
    a = 1
    b = 5
    baz( 5 )
end procedure foo

procedure foo2()
    use bar, only: a, b => c
    a = 2
    c = 3
end procedure foo2
```

Interfaces (prototypes)

Without a name, the `interface ... end interface` lists a bunch of procedures, in much the way a C header file lists function prototypes. Note that modules automatically create interfaces, so you need explicit interfaces only for external procedures (e.g. libraries, particularly Fortran 77 and other languages), and for procedures defined outside a module. In particular, you should write interfaces for any procedures defined in the same file as your main program. Thus it behooves you to write everything inside modules.

```
interface

    real function foo( arg1, arg2 )
        implicit none
        integer:: arg1
        real:: arg2
    end function

    subroutine bar( arg1 )
        implicit none
        integer:: arg1
    end subroutine

end interface
```

Interfaces (overloading generic procedures)

Generic (or overloaded) procedures are also defined using interface. Specify the generic name for the procedure, then all the interfaces for specific procedures. If a specific procedure is defined in a module, just use 'module procedure name' instead of repeating the interface.

This code defines a generic function `norm` which can be applied to matrices, and real and complex vectors. The function matrix_norm has been defined in a library, instead of being in another module like norm_complex.

```
module norm_m
    use complex_m, only: norm_complex

    interface norm
        interface
            real function matrix_norm( A )
                real:: A(:,:)
            end subroutine
        end interface

        module procedure norm_complex, norm_real
    end interface norm

contains

    real function norm_real( x )
        real:: x(:)
        ...code...
    end subroutine
end module norm_m


module complex_m
contains
    real function norm_complex( x )
        complex:: x(:)
        ...code...
    end subroutine
end module norm_complex_m
```

The procedures must be differentiated by their argument types, both by position and by name. So the following won't work, because `norm(A=X, kind='2')` matches both foo1 and foo2.

```
    interface norm
        interface
            subroutine foo1( A, kind )
                real:: A(:,:)
                character(len=1):: kind
            end subroutine
        end interface


        interface
            subroutine foo2( kind, A )
                real:: A(:,:)
                character(len=1):: kind
            end subroutine
        end interface
    end interface
```

To overload in C++, you define two procedures with the same name, instead of mapping different procedures to a common name. The equivalent in C++, assuming matrix and vector classes, might be:

```
    float norm( matrix& A ) { ... }
    float norm( vector& x ) { ... }
    float norm( complex_vector& x ) { ... }
```

The C++ syntax has several advantages. It is more concise. You can easily overload procedures that are already overloaded. C++ also has templates that parameterize a function by data type. In Fortran you have to repeat the code for each data type.

Input and Output

List directed input

```
    read *, variables
```

For list directed input, the data is delimited into fields by comma, whitespace, / slash, or newline. Strings are delimited by either `'...'` single or `"..."` double quotes, or are a single "word". Two consecutive commas are a null value---the variable remains unchanged. Slash terminates input---all remaining variables are unchanged.

User-formatted input

```
    read '(formats)', variables
```

or

```
    read label, variables
    label  format (formats)
```

User-formatted input uses fixed width fields, specified in formats, which is a comma separated list of the format codes below.

Format	Description
`Iw`	Read w characters as an integer
`Fw.d`	Read w characters as a real. If it doesn't contain a decimal point, assume d decimal places.
`Ew.d`	For input, same as Fw.d
`Aw`	Read w characters as string. If w > len(variable) then it truncates the **left** side of text. (This is because formatted output is right aligned, so the leading spaces should be lopped off.)
`A`	Read w=len(variable) characters as a string.
`Lw`	Read w characters as logical. Text starting with T or .T is true, text starting with F or .F is false.
`wX`	Ignore w characters.
`Tc`	Move ("tab") to absolute position c
`TLw`	Move left w characters
`TRw`	Move right w characters
`/`	Ignore rest of line (record) and read next line (record)

List directed output

List directed output uses default formats to format each expression. The format, particularly field widths for different types, is compiler dependent.

```
    print *, expressions
```

User-formatted output

```
    print '(formats)', expressions
```

or

```
    print label, expressions
    label format (formats)
```

User-formatted output uses fixed width fields, specified in formats, which is a comma separated list of the format codes below. All fields are right aligned. Unlike C's printf, if a value doesn't fit in the field, it is printed as **** asterisks, instead of expanding the width.

Format	Description
`Iw`	Print integer in w characters
`Fw.d`	Print real in w characters with d decimal places
`Ew.d`	Print real in exponent format, in w characters with d decimal places in mantissa and 4 characters for exponent
`Aw`	Print string in w characters
`A`	Print string in len(variable) characters
`Lw`	Print w-1 blanks, then T or F for logical
`wX`	Print w blanks
`Tc`	Move ("tab") to absolute position c
`TLw`	Move left w characters
`TRw`	Move right w characters
`/`	Start new line

An array name, or array slice, refers to the whole array or slice. So this reads 5 reals into the first row of a matrix, then prints out the first row:

```
    real:: A(5,5)
    read  *, A(1,:)
    print *, A(1,:)
```

File input/output

```
    open ( unit=n, file=filename, status=file_status, iostat=variable )
    read ( unit=n, fmt=format,    iostat=variable ) variables
    write( unit=n, fmt=format,    iostat=variable ) expressions
    close( unit=n )
```

status is "old" -- must already exist "new" -- must not exist (like O_EXCL?) "replace" -- delete if exists (like O_TRUNC) "scratch" -- unnamed temporary, delete when program exits "unknown" -- compiler dependent; usually like old if it exists, like new if it doesn't exist action is read write readwrite form is formatted -- e.g. text unformatted -- e.g. binary position is rewind -- e.g. beginning append -- e.g. end asis -- don't modify position if unit is already open (eek!)

n is unit number, or * for default input/output unit. format is '(formats)' or label, or * for list-directed input/output. iostat saves error code to the given variable.

backspace unit backspace (unit=n, iostat=var) -- position before previous record (e.g. to re-read a record) rewind unit rewind(unit=n, iostat=var) -- go to beginning

Binary unformatted input/output

Omit the format specifier. Writes data in a binary format. Must be read back with exactly the same types of variables, on the same computer architecture!! (E.g. with the same endianness and word size.) Example:

```
    write( unit=9 ) x, y, z
    read( unit=9 ) a, b, c
```

Intrinsic Functions

I know I said I disliked having a list of intrinsics with no explanation, but I have to start somewhere. This list is initially copied from the Queen's University of Belfast intro, but has since been extensively modified.

In the following intrinsic function definitions arguments are usually named according to their types (I for integer C for character, etc.), including those detained below. Optional arguments are shown in square brackets [ ], and keywords for the argument names are those given.

Uncategorized	
mvbits(from, frompos, len, to topos)	copy bits.
kind(x)	returns an integer describing the kind. (e.g. real, double precision, etc.)
set	a string containing a set of characters.
back	a logical used to determine the direction a string is to be searched.
mask	a logical array used to identfy those element which are to take part in the desired operation.
dim	a selected dimension of an argument (an integer).
Optional argument query	
present(x)	true if argument x is present.
Conversion functions	
aimag(z)	return the imaginary part of complex number z.
cmplx(x[, y][, kind])	converts to a complex number with real part x and imaginary part y.
dble(x)	converts x to a double precision real. (Use real(x,kind) instead.)
real(x[, kind])	converts x to a real.
logical(l[, kind])	convert between different logical kinds.
Rounding functions	
int(x[, kind])	returns x rounded toward 0; returns integer. Same as in C
nint(x[, kind])	returns x rounded to nearest; returns integer. (Like lrint in C.)
aint(x[, kind])	returns x rounded toward 0; returns real.
anint(x[, kind])	returns x rounded to nearest; returns real. (Like rint in C.)
ceiling(x)	returns x rounded up; returns integer. (In C, ceil returns float.)
floor(x)	returns x rounded down; returns integer. (In C, floor returns float.)

Examples

x	**1.75**	**1.25**	**-1.25**	**-1.75**
int(x)	1	1	-1	-1
nint(x)	2	1	-1	-2
aint(x)	1.00	1.00	-1.00	-1.00
anint(x)	2.00	1.00	-1.00	-2.00
ceiling(x)	2	2	-1	-1
floor(x)	1	1	-2	-2
exponent(x)	1	1	1	1
fraction(x)	0.875	0.625	-0.625	-0.875
x	**7.00**	**3.50**	**1.75**	**0.875**
exponent(x)	3	2	1	0
fraction(x)	0.875	0.875	0.875	0.875

Math functions	
abs(x)	absolute value.
conjg(z)	complex conjugate.
dim(x, y)	maximum of x-y or 0.
dprod(x, y)	double precision product.
exp(x)	exponential.
log(x)	natural logarithm.
log10(x)	base 10 logarithm.
max(x1, x2[, x3...])	returns the maximum value.
min(x1, x2[, x3...])	returns the minimum value.
mod(x, p)	remainder modulo p, i.e. x-int(x/p)*p.
modulo(x, p)	x modulo p.
sign(x, y)	absolute value of x times the sign of y.
sqrt(x)	square root.
Trig functions	
acos(x)	inverse cosine.
asin(x)	inverse sine.
atan(x)	inverse tan.
atan2(x, y)	inverse tan.
cos(x)	cosine.
cosh(x)	hyperbolic cosine.
sin(x)	sine.
sinh(x)	hyperbolic sine.
tan(x)	tan.
tanh(x)	hyperbolic tan.

Random number	
random_number(x)	fill in x with random numbers in the range [0,1]. x may be a scalar or array.
random_seed([size][, put][, get])	initialise or reset the random number generator.
Floating point manipulation functions	
exponent(x)	returns the exponent part of x.
fraction(x)	returns the fractional part of x, such that x = fraction(x) * radix(x)**exponent(x). (This is *not* x - int(x).)
nearest(x, s)	returns the nearest different machine specific number in the direction given by the sign of s.
rrspacing(x)	returns the reciprocal of the relative spacing of model numbers near x.
scale(x)	multiple x by its base to power i.
set_exponent(x, i)	sets the expontent part of x to be i.
spacing(x)	returns the absolute spacing of model numbers near x.
Vector and matrix functions	
dot_product(vector_a, vector_b)	returns the dot product of two vectors (rank one arrays).
matmul(matrix_a, matrix_b)	returns the product of two matrices, or matrix and vector.
Array reduction functions	
all(mask[, dim])	returns .true. if all elements of mask are .true.
any(mask[, dim])	returns .true. if any elements of mask are .true.
count(mask[, dim])	returns the number of elements of mask that are .true.
maxval(array[, dim][, mask])	returns the value of the maximum array element.
minval(array[, dim][, mask])	returns the value of the minimum array element.
product(array[, dim][, mask])	returns the product of array elements
sum(array[, dim][, mask])	returns the sum of array elements.
Array enquiry functions	
allocated(array)	returns .true. if array is allocated.
lbound(array[, dim])	returns the lower bounds of the array.
shape(source)	returns the array (or scalar) shape.
size(array[, dim])	returns the total number of elements in an array.
ubound(array[, dim])	returns the upper bounds of the array.
Array constructor functions	
merge(tsource, fsource, mask)	returns value(s) of tsource when mask is .true. and fsource otherwise.
pack(array, mask[, vector])	pack elements of array corresponding to true elements of mask into a rank one result
spread(source, dim, ncopies)	returns an array of rank one greater than source containing ncopies of source.
unpack(vector, mask, field)	unpack elements of vector corresponding to true elements of mask.
Array reshape and manipulation functions	
cshift(array, shift[, dim])	performs a circular shift.
eoshift(array, shift[, boundary][, dim])	performs an end-off shift.
maxloc(array[, mask])	returns the location of the maximum element.
minloc(array[, mask])	returns the location of the minimum element.
reshape(source, shape[, pad][, order])	rehapes source to shape shape
transpose(matrix)	transpose a matrix (rank two array).
Numeric enquiry functions	
digits(x)	returns the number of significant digits in the model.
epsilon(x)	returns the smallest value such that real(1.0, kind(x)) + epsilon(x) is not equal to real(1.0, kind(x)).
huge(x)	returns the largest number in the model.
maxexponent(x)	returns the maximum exponent value in the model.
minexponent(x)	returns the minimum exponent value in the model.
precision(x)	returns the decimal precision.
radix(x)	returns the base of the model.
range(x)	returns the decimal exponent range.
tiny(x)	returns the smallest positive number in the model.
Character functions	
achar(i)	returns the ith character in the ascii character set (like C's chr).
char(i[, kind])	returns the ith character in the machine specific character set (like C's chr).
iachar(c)	returns the position of the character in the ascii character set (like C's ord).
ichar(c)	returns the position of the character in the machine specific character set (like C's ord).
Character functions -- whitespace	
adjustl(string)	adjusts string left by removing any leading blanks and inserting trailing blanks.
adjustr(string)	adjusts string right by removing trailing blanks and inserting leading blanks.
trim(string)	removes trailing spaces from a string.
Character functions -- length	
len(string)	returns the length of a string.
len_trim(string)	returns the length of a string without trailing blanks.
Character functions -- comparison	
lge(string_a, string_b)	lexically greater than or equal to.
lgt(strin_a1, string_b)	lexically greater than.
lle(string_a, string_b)	lexically less than or equal to.
llt(string_a, string_b)	lexically less than.
Character functions -- concat	
repeat(string, ncopies)	concatenates ncopies of string.

Character functions – find	
index(string, substring[, back])	returns the leftmost (rightmost if back is .true.) starting position of substring within string.
scan(string, set[, back])	returns the index of the leftmost character of string that belongs to set (rightmost if back is .true.), or 0 if none belong.
verify(string, set[, back])	(same as scan?) returns zero if all characters in the string belong to the set, otherwise returns the index of the leftmost character that is not in the set (rightmost if back is .true.).
Kind functions	
kind(x)	returns the kind type parameter value.
selected_int_kind(r)	kind of type parameter for specified exponent range.
selected_real_kind([p][, r])	kind of type parameter for specified precision and exponent range.
Bit enquiry functions	
bit_size(i)	returns the number of bits in the model.
Bit manipulation functions	
btest(i, pos)	is .true. if bit pos of integer i has a value 1.
iand(i, j)	logical .and. on the bits of integers i and j.
ibclr(i, pos)	clears bit pos of interger i to 0.
ibits(i, pos, len)	extracts a sequence of bits length len from integer i starting at pos
ibset(i, pos)	sets bit pos of integer i to 1.
ieor(i, j)	performas an exclusive .or. on the bits of integers i and j.
ior(i, j)	performes an inclusive .or. on the bits of integers i and j.
ishift(i, shift)	logical shift of the bits.
ishiftc(i, shift[, size])	logical circular shift on a set of bits on the right.
not(i)	logical complement on the bits.
Transfer functions	
transfer(source, mold[, size])	converts source to the type of mold.
Pointer association status enquiry functions	
associated(pointer[, target])	returns .true. if pointer is associated.
Date and time	
date_and_time([date][, time][, zone][, values])	real time clock reading date and time.
system_clock([count][, count_rate][, count_max])	integer data from the real time clock.

Wishlist

These are features I sorely miss from other languages, or features that really annoy me in Fortran.

- Allow declaring variables inline, instead of at top of procedure, as in C++ and C99.
- Operators +=, –=, *=, /=. (Unfortunately they used /= for .ne.! Dumb heads.)
- Allow implicit line continuation inside parenthesis ( ), as in Python. This eliminates 98% of the need for & line continuation.
- Make `implicit none` always active in free form, or at least the default in free form. In gfortran this can be done with `-fimplicit-none`
- Overload == and /= for logical expressions. (Same as .eqv. and .neqv.)
- Define .and. and .or. as short-circuiting. The Fortran 77 standard says they can be short-circuiting, but the wording is a bit vague, so might allow the compiler to evaluate the second exrpession even when it isn't needed.

- Better IO functions. In particular, for unformatted IO, instead of assuming a rigid fixed column record format. For instance,

  ```
  read '(i4)', var
  ```

  breaks if the input is a 5 digit integer, whereas in C,

  ```
  scanf( '%d', &var );
  ```

  slurps up all available digits. Similarly write fails (prints **** asterisks) for an unexpectedly large value, while printf extends the width.
- Defined constants for error codes.
- Add tab to the Fortran character set, so I don't have to set `-Wtabs`. In my opinion, tabs are better than spaces for indentation, because they are always consistent, and each user can define them to be as wide as they like. With spaces, it's easy to inconsistently indent some lines by, say, 3 spaces and others by 4 spaces.
- Namespaces. Some way to import a module without import all its identifiers, and then syntax to access the module's identifiers, like C++'s namespace::identifier syntax. The `use foo` syntax pollutes the current namespace, and any name conflicts have to be manually resolved by renaming. For instance, this generates an error:

  ```
  program test
      use foo
      use bar
      print *, a
  end program test

  module foo
      integer:: a=1
  end module

  module bar
      integer:: a=2
  end module
  ```

  It can be fixed with

  ```
  program test
  ```

```
        use foo, foo_a => a
        use bar, bar_a => a
        print *, foo_a, bar_a
    end program test
```

but a cleaner solution would be extending % to work with modules as

```
    program test
        use foo
        use bar
        print *, foo%a, bar%a
    end program test
```

- Initialize arguments without implicitly making them static. See bugs.
- Default values for arguments. The optional thing makes it rather complicated. For instance,

```
    subroutine symmetric_solve( A, b, uplo='u' )
        real:: A(:,:), b(:)
        character(len=1):: uplo
        ...code...
    end subroutine
```

is easier to write and to understand than

```
    subroutine symmetric_solve( A, b, uplo )
        real:: A(:,:), b(:)
        character(len=1), optional:: uplo
        character(len=1):: local_uplo
        local_uplo = 'u'
        if ( present(uplo)) local_uplo = uplo
        ...code...
    end subroutine
```

- Fortran apparently can't promote the types of any variables when trying to match an overloaded function. So this won't work: call la_symv('u', 6, 1d0, B, 6, d, 1, 0d0, Bd, 1) you have to explicitly use the correct types (real or double): call la_symv('u', 6, 1.0, B, 6, d, 1, 0.0, Bd, 1) call la_symv('u', 6, 1d0, B, 6, d, 1, 0d0, Bd, 1) but that defeats the ability of parameterizing the precision with real(wp). We're forced to use constants one = real(1,wp) and zero = real(0,wp).