Programming in Fortran

Youjun Hu

Institute of Plasma Physics, Chinese Academy of Sciences
Email: yjhu@ipp.cas.cn

As a computational physicist using Fortran for decades, I summarize what I know about Fortran and what features are frequently used in practical programming.

1. Why Fortran?

Fortran is a language tailored to the specific task of numerical computation. Most programming languages will have a flavor of Fortran if they are used to perform numerical computation. Fundamental support for multi-dimensional arrays in Fortran makes it convenient to write numerical programs, in which arrays are often the only data structure needed. Practical reasons why a computational physicist should learn Fortran include (1) Fortran is dominant in parallel computing (only C can compete with Fortran in this regard since MPI only supports C and Fortran). Fortran programs are using most of the resources of the largest supercomputers in the world. This is a strong indication that Fortran is dominant in the high performance computing area; (2) Fortran is still the most widely used programming language in the computational physics community so that one can not totally avoid reading or revising Fortran codes if one wants to communicate with people in the computational physics community.

2. Compiler options help find bugs

Compilers are our friends, helping us find bugs earlier.

For debug purpose, I primarily use open source compiler `gfortran`. The compiler options mentioned below will be for `gfortran` by default. The following are some useful `gfortran` compiler options that can help find bugs:

```
-std=f2008 -fcheck=all -Wall  -Wextra -fbounds-check -fimplicit-none
```

3. Basic syntax

Fortran is case-insensitive (this is different from most popular languages).

Use either newline or semicolon to complete an statement (this is different from C which neglects newline). For example:

```
int i
real j
```

is equivalent to

```
int i; real j
```

Use exclamation mark (`!`) for one line comment, which tells the compiler to discard everything from where the mark is found up to the end of the same line. Fortran does not have a keyword for defining block comment (multi-line comment) similar to C (`/* */`).

4. Program units

The recommended way of organizing Fortran code is to use only two top units: `program` and `module`. And place `function`/`subroutine` in `modules`, which makes the full interfaces of the procedures available to a calling unit and thus makes it possible for compilers to check argument match at compiling time. This helps to find all bugs related to argument mismatch before actually running the code.

4.1. Main program

```
program your_name
  use your_modules
  implicit none
  real :: x
  x = 1.0 !assignment, this is a comment
  call some_subroutine
end program your_name
```

4.2. Functions and subroutines

Define subroutines:

```
subroutine p(arg1, arg2, more_arguments)
  some codes here
end subroutine p
```

Define functions:

```
function myfunc(arg1, arg2, more_arguments) result (myval)
  some codes here
end function myfunc
```

Use keyword `call` to invoke a subroutine:

```
call p(arg1, arg2, more_arguments)
```

Functions can be used in all expressions, e.g.,

```
a = myfunc(1,2)*1.0
```

Fortran pass arguments by reference (not by values). A subroutine or function knows the **memory location** of the actual arguments passed to them via the argument list. Generally no copying in and copying out of an array, but sometimes, copying in and copying out may be involved.

4.3. Module

A module contains specifications and definitions that can be accessed from other program units. These definitions include data object definitions, namelist groups, derived-type definitions, procedure definitions, and procedure interface blocks. For example:

```
module mod_name
  use some_module
  implicit none
  real:: x
contains !the following must be subroutine/function
  subroutine p(arg1,...)
     some_codes_here
```

```
  end subroutine p
end module mod_name
```

As mentioned above, placing `function`/`subroutine` in `modules` (by using keyword `contains`) can make the full interfaces of the procedures available to a calling unit and thus makes it possible for compilers to check argument match at compiling time. This helps to find all bugs related to argument mismatch before actually running the code.

All variables in a modules are assumed by most fortran compilers to have `save` attribute, but this is not required by the fortran standard. One way to make sure that all variables in a module retain their values throughout the runtime is to `use` that module in the main program, so that the module never goes out of scope and thus the values of the variables are retained throughout the runtime.

5. Control flow

5.1. IF CONSTRUCT

```
if (logical_expression1) then
   some_codes
elseif (logical_expression2) then
   some_codes
else
   some_codes
endif
```

where `elseif` and `else` clauses can be absent. A single branch `if` construct can be simplified as

```
if (logical expr)  one_statement
```

Table 1 summarizes the relation operators and logical operators used in forming logical expressions:

Operator	Syntax 1	Syntax 2
Equals	`a == b`	`a .eq. b`
Not equals	`a /= b`	`a .ne. b`
Less than	`a < b`	`a .lt. b`
Less than or equal to	`a <= b`	`a .le. b`
Greater than	`a > b`	`a .gt. b`
Greater than or equal to	`a >= b`	`a .ge. b`
Logical and		`c .and. d`
Logical or		`c .or. d`
Logical negation		`.not. c`

Table 1. Relation and logical operators in Fortran. Here `a` and `b` are of number type; `c` and `d` are of logical type.

5.2. Loop structure

Fortran `Do` loop:

```
do i = nstart, nfinal, stride
   print *, 'i=', i
enddo
```

If you change the value of the loop index `i` inside the above loop structure, e.g. `i=i+1`, the compiler will complain:

```
Error: Variable 'i' at (1) cannot be redefined inside loop beginning at (2)
```

The above loop is similar to the `for` loop in C. For comparison, consider the following `for` loop of C:

```
  for(j=0;j<10;j=j+1){
    printf("hello,%d\n",j);
    j=j+1;  }
```

where we also modify the value of loop index variable `j` inside the loop and this is allowed by C compilers. This comparison indicates that C is flexible and, as a side effect, more prone to possible bugs.

Another loop structure using `while`:

```
do while(i<10)
   write(*,*) 'i=',i
   i=i+1
enddo
```

The above `while` loop can also be implemented by using `if` and `goto`:

```
10 if (i<10) then
   write(*,*) 'i=',i
   i=i+1
   goto 10
endif
```

The `while` loop can also be implemented by using `do` and `exit`:

```
do
   write(*,*) 'i=',i
   i=i+1
   if(i>=10) exit
enddo
```

Here the `exit` statement will transfer control outside the `do` loop before the `enddo` is reached. After an `exit` statement has been executed, control is passed to the first statement after the loop. In passing, we note that there is a `cycle` statement that transfer control back to the beginning of the loop to allow the next iteration of the loop to begin.

6. Basic data type

Fortran is a statically typed language, which means that the type of values that a variable can take is fixed at compile time and does not change during runtime. Fortran does not have the type inference capability available in some languages such as haskell, which means that we need to manually declare the type of all variables used in a program. The `implicit` rule that enables us to omit type declaration is "syntactic sugar" that we should avoid using in practice because it is bug-prone. To totally avoid the problematic "implicit rules" of fortran, using either `implicit none` or the compiler option `-fimplicit-none`, or using both for safety.

Fortran has five basic data types, namely `integer`, `real`, `complex`, `logical` and `character`.

Type	Example values
integer	`1, 0, -2`
real	`1.25`, 2.1e8
complex	`(1.0, 2.0)`
character	`''hello''`
logical	`.true. .false.`

Table 2. Built-in types of Fortran.

Each data type has some attributes that can be further specified. For example, the length of a `character` variable is an attribute, which can be specified by using

```
character (len = 40) :: name
```

Without specifying the value of `len` parameter, the length of the variable is default to one.

To specify the number of bytes for storing a `real` variable, we can use the `kind` parameter:

```
real(kind=4) :: xs    ! 4  byte float
real(kind=8) :: xd    ! 8  byte float
real(kind=16) :: xq   ! 16 byte float
```

However the meaning of the value of the `kind` parameter depends on the compiler. For example, some compilers may use the following scheme:

```
real(kin=1) :: xs    ! 4  byte float
real(kind=2) :: xd   ! 8  byte float
real(kind=3) :: xq   ! 16 byte float
```

I used to use

```
integer,parameter:: p_=kind(1.0d0) !return the kind value of a double precison constant
real(kind=p_) :: xd !use this value as the kind value
```

to define a double-precision variable (8 byte float). However, a more convenient way is NOT to specify the `kind` value of `real` variable in source code and use compiler options to specify the precision we want. For example, we write in the code: `real :: abc` and then compile the code by using the compiling option `-fdefault-real-8` (for `gfortran`) to specify a 8 byte float number. For `ifort`, the corresponding option is `-r8`.

7. Array data structure

Scientific computing often deals with simple data structures with homogeneous elements, such as arrays, which are often the only data structure used in a numerical code. This makes derived types not so frequently needed. We first discuss arrays. Derived types are discussed later.

Use `dimension` to declare arrays. For example:

```
real, dimension(0:2, 1:4, -1:3) :: A
```

defines a `real`-type array `A` with **dimensionality** being 3; the **lower bound** and **upper bound** of the 1st dimension are `0` and `2`, respectively; the **extent** along each dimension is equal to `upper_bound-lower_bound+1`, and for this case, is 3, 4, and 5, respectively. The dimensionality of an array is often called as "**rank**" in Fortran (although this name can be confused with the rank of a matrix in linear algebra, which corresponds to the maximal number of linearly independent columns of a matrix).

The combination of the **rank** and corresponding **extents**, is called the **shape** of an array. In the above case the shape of the array is `(3; 4; 5)`. Two arrays are conforming with each other if they have the same shape.

The following is a shortcut of defining an array (which omits the `dimension` keyword):

```
real :: A(0:2, 1:4, 2:6)
```

We need specify **bounds** in each dimension in order to completely specify a multidimensional array. If we do not specify the **lower bound** of the array, lower bound of array in Fortran is by default 1. The following declaration:

```
real :: B(1:3, 1:4, 1:6)
```

can be simplified as

```
real :: B(3, 4, 6)
```

In summary:

- **Rank** is the number of dimensionality of an array
- **Extent** is the number of elements along a dimension
- **Bound** is the upper or lower index along a dimension
- **Shape** is the combination of **rank** and **extents**.

7.1. ACCESS ARRAY ELEMENTS

Using index to access single element of an array. For example

```
B(1,2,1) = 1.0
```

Use the index range (start:end) to access multiple elements of an array (sub-array or array-slice). For example:

```
write(*,*) B(1:2, 1:3, 3)
```

When referring to an array slice, both lower bound and upper bound can be omitted and they defaul to the declared array bounds . For example:

```
write(*,*) B(:, :, 3)
```

7.2. MEMORY LAYOUT OF MULTI-DIMENSIONAL ARRAY

Since memory is laid out in one-dimension (each location has a single address index, not a set of indices), a compiler must decide how to map multi-dimensional arrays to memory locations. Different language use different conventions. In Fortran, arrays are stored "by column" in memory, which means putting each column of a 2D matrix in contiguous memory, as is illustrated in Fig. 1.

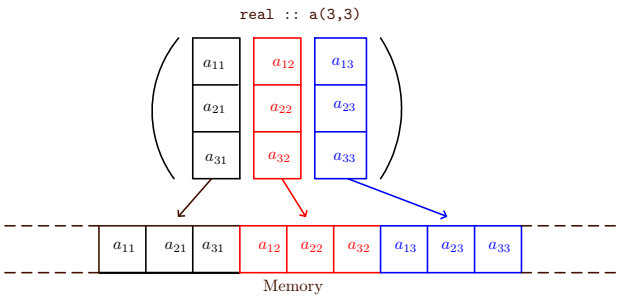

Figure 1. Fortran arrays are stored in memory "by column". This layout is often called "column-major order". As a comparison, C and Python use the "row-major order" layout, which puts each row of a matrix in contiguous memory.

The above illustration is for 2D array. It is obvious how to generalize the "column-major order" layout to higher dimensional arrays: when traversing memory continuously, the first index of an array is the fastest changing, the second index is the second fastest changing, and so on.

Why do we care the memory layout of an array?

Firstly, knowing the layout enables us to calculate the memory offset of an array element. For example, for an array declared with `real(kind=8) :: a(m,n)`, the memory distance of an element `a(i,j)` from `a(1,1)` is $[m \times (j-1) + i - 1] \times 8$byte if the layout is the column-major order. The distance would be $[n \times (i-1) + j - 1] \times 8$byte if the layout was the row-major order.

Secondly, the array layout is important to CPU cache and thus to code performance. A CPU views the memory as in 1D layout. When a CPU fetches an element from memory to operate on, it will guess that future operations may need some elements near that memory location. Therefore, a CPU will load a contiguous memory block near that location to CPU cache in order to make it comparatively quick to access elements potentially needed in future operations.

Thirdly, **vectorized instructions** of modern CPUs also requires consecutive access. (The vector processing can either be done with intrinsics, syntax hints of a Language, or by relying on the compiler's auto-vectorizer.)

So, to get good code performance, always traverse data in the order they are laid out in memory.

Layout of multi-dimensional arrays in memory is just a logical abstraction of the 1D linear memory. Different choices of layout can be simply considered as personal preference of the designers of a language. The personal preference of a language designer is related to the data structure and the type of operation on the data frequently used in his/her routine work. If we try, we can always find some reasons why a language prefers one layout over the another. For Fortran, I believe the reason is related to multiplication of a matrix with a vector, which can be arranged as

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = b_1 \begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \end{pmatrix} + b_2 \begin{pmatrix} a_{12} \\ a_{22} \\ a_{32} \end{pmatrix} + b_3 \begin{pmatrix} a_{13} \\ a_{23} \\ a_{33} \end{pmatrix}$$

In this form, it is an operation over individual columns of the matrix. Then it is beneficial to store a column in contiguous memory.

For C and Python, which choose the "row-major order" layout, I believe the reason is related to the view that a multi-dimensional array is considered as a nested 1D array. Using Python list as an example:

```
In [3]: mylist = [[1,2,3], [4,5,6]]
```

which is a nested list corresponding a 2×3 matrix. Since the first row of the matrix corresponds to a 1D sub-list `mylist[0]`, it is natural to store these elements in contiguous memory (list elements are not necessary in continuous memory, here we assume that a 1D list is a 1D array).

7.3. Performance improvement due to the spatial locality in memory access patterns

Let us see a real example in Fortran, which clearly demonstrates the performance improvement thanks to the spatial locality in memory access patterns:

```
program main
  implicit none
  integer, parameter:: m=10000, n=30000
  real :: a(m,n), b(n), c1(m), c2(m)
  integer :: i, j
  real:: tarray(3) !record the cputime
  CALL RANDOM_NUMBER(a)
  CALL RANDOM_NUMBER(b)
  call cpu_time(tarray(1)) !f95 intrinsic subroutine returning the cpu clock
```

```
  c1=0.0
  do i=1,m !switching the two loops can improve the efficiency
     do j=1,n
        c1(i)=c1(i)+a(i,j)*b(j)
     enddo
  enddo
  call cpu_time(tarray(2))
  write (*,*) 'CPU time used (seconds) in inefficent algorithm', tarray(2)-tarray(1)

  c2=0.0
  do j=1,n
     c2(:)=c2(:)+a(:,j)*b(j)
  enddo
  call cpu_time(tarray(3))
  write (*,*) 'CPU time used (seconds) in efficient algorithm', tarray(3)-tarray(2)
end program main
```

The above code multiplies a matrix by a vector. Two methods are implemented, one is the naive way, another is using operation over individual columns of the matrix, as discussed above. Compile and run the code:

```
$ gfortran  c_r.f90  && ./a.out
 CPU time used (seconds) in inefficent algorithm   1.08469510
 CPU time used (seconds) in efficient algorithm  0.208189011
```

There is a factor of 5 performance gain due to the spatial locality in memory access patterns. In this example, the spatial locality help both cache and vectorized instructions, since we use Fortran array slice syntax, which usually enables vector processing. In the naive method, if we switch the order of the two loops, then we obtain spatial locality in memory access patterns, but the vector processing may not be enabled because we do not provide sufficient syntax hints to the compiler. In this case, the result is as follows:

```
$ gfortran  c_r.f90  && ./a.out
 CPU time used (seconds) in inefficent algorithm  0.621351004
 CPU time used (seconds) in efficient algorithm  0.207148075
```

The performance of the naive method improves but still 3 factor slower than the vectorized version. Let us add `-O3` optimization, which may try to vectorize any codes than can be vectorized. The result is as follows:

```
$ gfortran -O3 c_r.f90  && ./a.out

CPU time used (seconds) in inefficent algorithm   7.00249672E-02
CPU time used (seconds) in efficient algorithm   7.00830221E-02
```

Finally, we obtain the same performance.

In passing, I would like to mention another important tip that can improve spatial locality of memory access and hence code performance: avoid using fine-grained derived datatypes or many small arrays, instead, try to gather them together into an array to improve locality in memory access. To demonstrate this, let us consider a famous example in particle simulations, where each particle has three spatial coordinates, say x, y, z. The intuitive method would use three arrays, `x(n),y(n),z(n)`, to store these coordinates for each particle, where `n` is the total particle number, which is usual a large number, e.g. 10^6. The jth elements of the three arrays will be frequently used together in a computation, for example `s=x(j)**2+y(j)**2+z(j)**2`. In this case, a better way of organizing the data is to combine the three arrays into one array, e.g., `cor(3,n)`, with the first index of the 1st dimension corresponding to x, second index to y, and third to z. Then the above expression would be `s=cor(1,j)**2+cor(2,j)**2+cor(3,j)**2`, where the three elements are contiguous in memory and thus can help the cache and speedup the computation.

7.4. Static arrays

We define static arrays as arrays whose shape and bounds are known at compile time, i.e., fully-declared-at-compile-time arrays.

If a static array is declared in a local scope, typically in a `subroutine/function`, its values are not retained between different calls to the `subroutine/function` (i.e., the array is automatically deallocated when the procedure returns, as automatic arrays (discussed later) would be). To retain the value, we can specify `save` attribute when declaring the array. Static arrays with `save` attribute are not allocated on the stack. They actually exist in neither the stack nor the heap. The are part of what's called the "data segment". Similar situation applies to the `static` variables in `C`.

Static arrays without `save` attribute are usually allocated on stack. However the size of stack is limited, large arrays have to be allocated in heap in order to avoid the stack overflow (this can be achieved by declaring an array as allocatable and allocating it manually, which usually imply that the allocation happens on the heap. However the Fortran standard has no concept of stack and heap, so this will be implementation (i.e. compiler) dependent.)

7.5. dynamic arrays

We define a dynamic array as an array whose extent along each dimension is not fully declared when they are defined. Partially declared extent includes two cases: (1) the extent is not specified, which corresponds to an allocatable array or (2) the extent is specified with variables, which corresponds to an automatic array.

The rank of a dynamic array is always known at compile time since the notation of array declariation does not allow un-specified rank.

Dynamic arrays are created at particular run times (by programmers explicitly or by compilers implicitly) with sizes determined by computed (or input) values. Fortran 90 has three varieties of dynamic arrays, namely:

- allocatable arrays
- automatic arrays
- pointer arrays

In high performance computing (HPC), it is usually recommended that we avoid using pointer arrays because they may prevent possible optimization by the compiler.

7.5.1. Allocatable arrays

Allocatable arrays are arrays whose bounds along each dimension are unknown at compile time (the rank is known), with their creation/deletion and their bounds/size being controlled by programmers at runtime. Use keyword `allocatable` to declare a allocatable array. In this case, only the rank needs to be specified. For example:

```
real, dimension(:,:), allocatable :: A
```

To allocate a dynamic array:

```
allocate(A(-1:3,4))
```

[Besides using `allocate` statement, there is another way in which an allocatable array will get (re-)allocated: assigning an array or array slice to an allocatable array name (rather than its slice). For example:

```
real, allocatable :: A(:,:)
```

```
real              :: B(3,4), C(5,6)
A = B  !A will get allocated, with the same shape and bounds as B
write(*,*) size(A,1), size(A,2), lbound(A,1)
A = C  !A will be de-allocated and then re-allocated to the same shape and bounds as C
write(*,*) size(A,1), size(A,2), lbound(A,1)
```

This is informally called "(re-)allocation on assignment", which makes dynamic arrays looks more like a variables in a dynamically typed language. However, there is overhead here because lots of checks/actions need to be done at runtime (e.g., is the allocatable array already allocated?, if allocated, is the shape and bounds are the same as the right-hand side array? if they differ, the allocated array will be de-allocated and then re-allocated to the same shape and bounds of the right-hand side).

Therefore, I usually try to avoid using this kind of automatic (re-)allocation on assignment. One can always suppress the (re-)allocation checks/actions of an array by using an array section (rather than an array name) in an assignment:

```
A(:,:) = B
```

In this case, if the allocatable array `A` is not allocated or already allocated but its shape does not match `B`, runtime error will appear.]

To de-allocate a dynamic array:

```
deallocate(A)
```

The creation and deletion of dynamic arrays are under full control of a programmer, with one exception that a dynamic array without `save` attribute will be automatically deleted if the declaration of the dynamic array goes out of scope. For example, an allocatable array without `save` attribute defined in a subroutine will be automatically deleted when the subroutine returns (i.e., the declaration of the allocatable array goes out of scope). The automatic deletion of dynamic array can be prevented by specifying `save` attribute when declaring the array. For most fortran compilers, dynamic arrays declared in a `module` are assumed to have `save` attribute by default.

7.5.2. Automatic arrays

An **automatic array** is a local array (not a dummy argument) of a `subroutine/function` and its definition involves variables whose values are unknown at compile time. Automatic arrays can not be delcared in `program` or `module` units.

This kind of arrays are called automatic arrays because their allocation (usually on stack) and de-allocation are automatically handled without explicit instructions from programmers. Specifically, a automatic array will be de-allocated automatically when it goes out of scope. We can not use `save` attribute to an automatic array. Otherwise, we will get compiling errors such as follows:

```
Error: Automatic object cannot have the SAVE attribute
```

If we really want to have a persitent arary, we should use **allocatable arrays** or **static arrays**, for which a `save` attribute is allowed. Note in passing that an allocatable array (not a dummy argument) defined in a `subroutine/function` without the `save` attribute will be automatically deallocated upon `subroutine/function` return, which is the same as an automatic array. (As a general rule, if an array/scalar is local to the routine, memory is allocated on entry to the routine and deallocated on return to the caller.)

Automatic arrays are usually allocated on the stack whereas allocatable arrays are allocated on the heap. But this behavior is not specified in the Fortran standard, and can depend on specific implementations. Does this difference between automatic arrays and allocatable arrays results in any difference in performance? (need to be examined)

Since automatic arrays can have different sizes between different calling to the `subroutine/function`, they are a kind of adjustable arrays.

In summary, automatic arrays are **local** arrays declared in a `subroutine/function/block` with definition involving variables whose values are unknown at compile time. Automatic arrays are not allowed to have the `save` attribute.

The specification of an automatic array is allowed to contain function calls, which can be either built-in or user-defined functions, with the only requirement that the functions must be `pure`, i.e., no side effects. One often used function in this case is the built-in inquiry function `size`, which is used to query the size of other arrays, and then the returned value is used in defining the automatic array. For example:

```
real :: myarray(size(a,1), size(a,2))
```

Local non-constant variables in a subroutine are not allowed to appear in the specification expression of automatic arrays. The reason is that an automatic arry needes to be allocated at the begining of the procedure, but a local variable does not have a defined value just on entry to the routine. Even if we initialize the value of the local variable when declaring it, this value is available only when the first executable statment in the procedure is executed because initialization is an assignment (an executable statement), which is not a pure function.

The specification expression of automatic arrys is allowed to use dummy arguments of the procedure and variables from modules (including its host module if the subroutine is contained in a module).

These rules are also applicable to the dummy argument arrays discussed in Sec. 7.6.

7.6. Dummy argument arrays

If a dummy argument of a procedure is declared as an array, then generally no array is actually allocated on entry to the procedure. It just refers to the actual argument arrays passed in from the calling unit. So dummy argument arrays are not new type of arrays, but the syntax of declaring them may be different from that of declaring static/automatic/allocatable arrays. And the syntax is used to determine what information of the actual argument array is passed in from the calling units. There are four syntaxes of declaring a dummy argument array:

```
real :: a(m,n) !explicit shape array
real :: b(m,*)   !assumed size array, not recomended now.
real :: c(2:,3:) !assumed shape array
real, allocatable :: d(:,:) !allocatable (dummy argument) array
```

Let us discuss these four types in turn.

7.6.1. explicit shape array

In the first two cases, i.e., `explicit shape array` and `assumed size array`, usually just a pointer to the first element of the actual argument array is passed in. Since only a pointer is available with no stride being available, the compiler has to assume that the stride is 1, i.e., the array is contiguous in memory. If we pass an array slice that is not contiguous in memory to a dummy argument declared as an `explicit shape array/assumed size array`, a temporary array will be created and the data of the array slice is copied in (and copied out when the subroutine returns) in order to guarantee that the resulting array is contiguous in memory so that the subroutine is doing right to the actual argument array. For this case, `gfortran` will give the following warning:

```
Fortran runtime warning: An array temporary was created for argument 'a' of procedure 'p'
```

If the dummy array is declared as an explicit shape array, rank mismatch between actual array and dummy array is allowed. However, this kind of use is error prone. Most time we prefer that the actual and dummy array have the same shape.

In the above example, the variables `m,n` appearing in the specification expression of the explicit shape array can be dummy argument variables or module variables, but can not be a local variable of the procedure. The following is an example of **explicit array**:

```
subroutine p(u,m)
 use some_module, only : n
 real :: u(m,n) !explicit shape array
end
```

A common bug when using explicit array is that the rank/extent/bound declared in a procedure are different from those of the actual argument array. As mentioned above, this mismatch is allowed by the compiler. If the mismatch is not intended by the programmer (i.e., the programmer made a mistake in the declaration), then the code usually will not work correctly.

7.6.2. ASSUMED SIZE ARRAY

An **assumed-size array** is an array that is a dummy argument and has an asterisk as the upper bound of the last dimension:

```
subroutine p( A, B, C )
 Integer :: A(*), B(5, *) !lower-bound of the last dimension default to 1
 Integer :: C(0:1, 2:*) !explicitly specify the lower-bound of the last dimension
end
```

The extent of the last dimension does not need to be specified because it is not needed (by the compiler) in calculating the memory address offset of an array element with respect to the first element. However, I prefer to use explicit-shape arrays over assumed-sized array since it is not a big deal to just omit a single bound. Omitting the last upper bound also makes bounds checking difficult since the last upper bound is not defined in the subroutine.

7.6.3. ASSUMED SHAPE ARRAY

To declare an argument as an `assumed shape array`, we only need to specify the rank and the lower bounds along each dimension (lower bounds default to 1 if not specified). The upper bounds will be automatically inferred during runtime from the length in each dimension of the actual array passed in. The following are examples of **assumed shape array**:

```
subroutine p1(A, B)
 real :: A(0:,2:) !upper bounds will be inferred from the length of the actual array
 real :: B(:,:)   !lower bounds are not specified and are default to 1
end
```

For the third case, i.e., `assumed shape array`, an array descriptor structure will be passed in, which contains the pointer to the first element of the actual argument array plus information on the length of each dimension and stride.

As is mentioned above, if the dummy array is declared as an explicit shape array and we pass in an array slice that is not contiguous in memory, an array temporary will be created and the data of the array slice is copied in and then copied out. On the other hand, if the dummy array is declared as an assumed shape array, no array temporary will be created. However, this does not mean the performance will be better than the case of using explicit shape array, where an temporary array is created and data be copied in and out. This is because the latter case can improve spatial locality in memory access pattern, although involving additional allocations and copying in and out operations. Therefore, which methods will win in terms of performance depends on specific applications.

If the dummy array is declared as an assumed shape array, the compiler will check agreement between the the rank of actual array and dummy array, giving error information if they mismatch. This kind of mismatch is allowed if explicit shape array is used. However, this kind of use is error prone. Most time we prefer that the actual and dummy array have the same shape.

In summary, assumed-shape arrays are dummy argument arrays where the extent of a dimension is inherited from the corresponding actual array. The dimensionality (i.e., rank) of the array is known and must be specified in the subroutine, for example, `a(:,:,:)` specifies that the rank is 3. If the actual array and the assumed shape array have different ranks, a compiler usually complains at compiling time, for example, `gfortran` raises the error information: `Rank mismatch in argument`.

Summary 2: we have two choices in declaring a dummy array in a subroutine. One choice is to declare it as an `explicit shape array` (using formal argument variables, or modules, or constants to specify the shape of the array). Another choice is to define `assumed (implicit) shape array`, for which we specify only the rank and lower bound (the lower bound is 1 if not specified) of the array and we do not specify the length along each dimension. The unspecified information will be inferred automatically from the actual argument array during runtime.

7.6.4. ALLOCATABLE DUMMY ARGUMENT ARRAY

For the fourth case, an dummy argument is declared as an allocatable array. In this case, the corresponding actual argument must also be an allocatable array, otherwise `gfortran` will raise `Error: Actual argument for ‘a’ must be ALLOCATABLE at (1).`

The actual argument allocatable array passed in can be already allocated, or not allocated yet and it can be allocated in the subroutine. One can use inquiry function `allocated()` to get the allocation status of an allocatable array passed in.

8. Pure function and subroutine

To define a pure function (functions with no side effects), we use the keyword `pure`. For example:

```
pure function f(x) result(z)
integer, intent(in):: x
integer :: z
z=x*2
end function f
```

All dummy argument variables of a pure function must be `intent(in)`. Pure subroutines can be defined in a similar way, except that the dummy argument variables can have `intent(out)`.

Since pure procedures do not allow side effects (i.e., do not change the state of the outer world), the compiler will raise error if we try to modify the value of a variable of a `module` that is used in this procedure:

```
Error: Variable ’tmp’ can not appear in a variable definition context (assignment) in PURE
procedure
```

Fortran pure functions can use module variables as (invisible) input, which can change between function invocations. Therefore fortran pure functions can not guarantee “same argument, same result”. This is different from the pure function concept in funtional programming languages, e.g. Haskell, where “same argument, same result” is guaranteed.

Local variables within the scope of a `pure` procedure cannot have the `save` attribute, which implies that they cannot be initialized when declared, or by a DATA statement.

Any procedures that are invoked from a `pure` procedure must be `pure`. No external I/O operations may occur within a PURE procedure since these change the stae of the outer world, i.e., side effects.

9. Elemental function

`Elemental` functions are pure functions defined with a single scalar dummy argument and a scalar return value, but they can be invoked with arrays as actual arguments, in which case the function will be applied element-wise, with a conforming array return value. This kind of function is indicated with the `elemental` prefix. Since it is implied that an elemental function is also pure, the `pure` prefix is not necessary.

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

Elemental subroutines may be defined in a similar way (`intent(out)` or `intent(inout)` arguments are allowed). The main benefit of elemental procedures is just for syntax convenience, it does not have semantics that lend themselves to effective parallelization.

10. Variable scope and lifetime

Not all variables are accessible from all parts of our program, and not all variables exist for the same amount of time. Where a variable is accessible and how long it exists depend on how it is defined. We call the part of a program where a variable is accessible its *scope*, and the duration for which the variable exists its *lifetime*.

10.1. Scope of variables: global and local variables

The scope of a variable, more accurately the scope of a naming binding (an association of a name to an entity) is the region of a computer program where the binding is valid: where the name can be used to refer to a value. Such a region is referred to as a scope block. In other parts of the program the name may refer to a different entity (it may have a different binding), or to nothing at all (it may be unbound).

The scope of a binding is also known as the visibility of an entity. In other words, which parts of your program can see or use it. In practice, for most programming languages, "part of a program" refers to "part of the source code", and is known as **lexical scope** (another kind of scope called **dynamic scope** will be discussed later). The lexical scope for a set of bindings in practice largely corresponds to a block, a function, or a file, depending on languages and types of entity.

Unlike C/C++, Fortran before 2008 standard does not have source code blocks where new variables can be defined. Fortran 2008 introduces the keyword `block` to define blocks, where local variables can be declared. For examples, the following code defines two nested blocks in which local variables `res1` and `res2` are defined.

```
PROGRAM foo
implicit none
  INTEGER :: a
  a=3.0
  write(*,*) 'a=',a
  BLOCK
    INTEGER :: res1
    res1 = a + 1
    write(*,*)  'res1=',res1
    BLOCK
      INTEGER :: res2
      res2 = res1 + 1
      write(*,*)  'res2=',res2,'res1=',res1
```

```
    ENDBLOCK
  ENDBLOCK
END PROGRAM foo
```

In summary, the `block` creates its own name space.

I use `block` to add temporary diagnostic codes to existing programs. Doing this way allow all temporary local variables to be gathered together near the diagnostic codes and thus easy to read and easy to remove. This is more convenient than adding and calling new external diagnostic subroutines. In passing, adding diagnostic internal subroutines using `contains` is as convenient as using `block` and may be more elegant.

In C/C++, variables defined outside of all functions are global variables. Fortran does not have such kind of global variables. To pass information between subroutines, besides using subroutine parameter lists, one way is to use variables in a `module` which can be shared between subroutines.

10.2. Local variables with `save` attribute

A local variable defined in a procedure can have the `save` attribute, which will make the local variable retain its value between different procedure calls. These variables are called persistent variables. For example:

```
function f()
integer, save :: i=1
i = i+1
print *, i
end function
```

However, for multiple-thread applications, this kind of local variable will be shared among different threads, which means that racing problem will appear if the procedure is called in multiple threads. Considering this situation, computer guys often say that this kind of local variables are "thread unsafe".

A local variables with `save` attribute is hidden input to the procedure in which the variable is defined.

10.3. Lifetime of variables

Variables with `save` attribute will be alive from their creation to the end of the program, i.e., they are not automatically destroyed when they go out of scope. As a result, if the variables with `save` attribute is a local variable defined in a subroutine, then its value will be retained between different revocations of the subroutine. In contrast, local variables without `save` attribute will be automatically destroyed when they go out of scope. As a result, their values can not be retained between different revocations of the subroutine.

10.4. Global variables

In C language, variables declared outside functions are global variables, which can be assessed from any functions. Fortran does not have this kind of global variables, but variables in `modules` are accessible to other program units if the modules are imported to that unit. In principle, a variable defined in a `module` is a local variable to that module, and will be destroyed when the program unit that uses the `module` returns. However, two situations (often appear in practice) can make the module variables not be destroyed during the entire runtime: (1) the `module` is used in the main program and thus never goes out of scope during the runtime, (2) `save` attribute is used for the variables in the `module`, so they retain their values. Most fortran compilers will treat variables defined in a `module` as with `save` attribute even if no `save` attribute is specified. But for safety, we should specify `save` attribute for variables in a `module` if we want their values to be retained when they go out of scope.

The above practice makes `module` variables look like global variables. For this reason, we often simply say that modules define global data.

Due to the global nature of `module` variables, they are shared across threads and are therefore thread-unsafe. To make an application thread-safe, we must declare the global data as THREADPRIVATE or THREADLOCAL.

11. Tips for improving performance

New syntaxes in a language seldom benefit performance. That is, "code with new syntax" doesn't imply "better compiler optimizations", and even F77 is a "high level" language. Efficient Fortran codes should stick to F77 features at their core for high performance, most of newer features would usually result in less efficient executables. F90 and newer would be mainly useful for readablity, maintainability and extensibility of the code

Do you have some strong evidence to support such statements?

Well, yes, there is evidence for this strategy. I do not claim, that all newer features have a negative impact on performance, or that you should not use them at all. Just in the very kernel, the F77 style is generally a good guide to produce efficient code. (Staying away from pointer and target attributes, do not use fine-grained derived datatypes, there was even a time when array syntax statements are slower than explicit do-loops, though compilers seem to have much improved). I am actually using F2003 features heavily, my point is just in the very kernel, there is probably little benefit.

Vectorization, if by this you mean the F90+ array syntax, is mostly a programmer convenience issue rather than allowing faster code. A competent compiler will vectorize the equivalent DO loop just as well.

There is one major reason why Fortran 77 programs might be faster: Allocated arrays (Fortran 90) are much slower than declared-at-compile-time arrays. Both are not put at the same place in memory (stack vs heap). The difference in memory management of stack and heap in Fortran can change the performance. The Fortran standard has no concept of stack and heap, so this will be implementation (i.e. compiler) dependent.

12. Coarray

13. Writing/reading files

After Fortran 2008, the recommended way of opening file is as follows:

```
integer :: myunit
open(newunit=myunit, file='myfile.txt')
write(myunit, *) 'hello world'   !use the unit number to write to the file
```

Here `newunit` is an `intent(out)` keyword argument, which returns an available unit number.

Appendix A. Object-oriented programming in Fortran

Object-oriented programming style is not often used in numerical codes. Here I discuss this just for my curiosity.

Fortran, the oldest high-level programming language, was designed and used mainly for the purpose of numerical computation. Although Fortran is also claimed as a general-purpose programming language, it lies behind other popular general-purpose Language, such as C/C++ and Python, in some aspects. One of these aspects is the supporting to the object-oriented programming.

Why is the object-oriented programming not widely adopted in scientific computing? Scientific computing often deal with simple data structures with homogeneous elements, such as arrays, which are often the only data structure used in a numerical code. This makes derived types not so frequently needed. Derived types are generalization of arrays and can contain inhomogeneous elements, which provide a more powerful organizing capability than arrays. Classes in object-oriented programming are a special kind of derived types that have procedures bound to.

Fortran's `module` can contain both data and procedures. But we can not use the name of a module to define an object. On the other hand, `module` can contain derived type definition and this derived type can be used to define an object. Therefore a Fortran `module` containing both derived type definitions and procedures can be used as the counterpart of `class` in object-oriented languages. However the binding between a derived type object and its procedures must be implemented manually by programmers before Fortran 2003. The following is a Fortran 95 example emulating object-oriented programming style.

```
module class_point
  implicit none
  type, public :: point
     real :: x
     real :: y
  end type point
contains
  function distance(this) result(z) !distance from origin
    type(point), intent(in) :: this
    real :: z
    z = sqrt((this%x)**2+(this%y)**2)
  end function distance

  subroutine add(this,another,my_sum)
    type(point), intent(in) :: this, another
    type(point):: my_sum
    my_sum%x=this%x+another%x
    my_sum%y=this%y+another%y
  end subroutine add

  subroutine print_point(this)
    class(point), intent(in) :: this
    real :: tmp
    tmp = distance(this)
    print *, 'Point: (x,y) = ', this%x, this%y, ' distance from orig = ', tmp
  end subroutine print_point
end module class_point

program point_test
  use class_point,only: point, distance,add, print_point
  implicit none
  type(point) :: p1,p2,p3    ! Declare variables of type point.
  p1 = point(3.0,4.0)   ! Use the implicit constructor
  call print_point(p1)       !pass the object to the procedure via a argument
  p2 = point(1.0,0.5)   ! Use the implicit constructor
  call add(p1,p2,p3)
  call print_point(p3)
end program point_test
```

The above is not real object-oriented programming because the procedures are not actually bound to the object. Fortran 2003 provides type-bound procedures: those procedures that appear in the definition of the derived type following the `contains` statement. The following is a revised version of the above program.

```
module class_point
  implicit none
  private !set the default as private
  type, public :: point
     real :: x
     real :: y
   contains
     procedure:: distance
     procedure:: printp
     procedure:: add
     procedure:: add_func
  end type point
contains
  function distance(this) result(z) !distance from origin
    class(point), intent(in) :: this
    real :: z
    z = sqrt((this%x)**2+(this%y)**2)
  end function distance

  subroutine add(this,another,my_sum)
    class(point), intent(in) :: this, another
    class(point):: my_sum

    my_sum%x=this%x+another%x
    my_sum%y=this%y+another%y

  end subroutine add

  function add_func(this,another)  result(my_sum)
    class(point), intent(in) :: this
    type(point), intent(in):: another
    type(point):: my_sum

    my_sum%x=this%x+another%x
    my_sum%y=this%y+another%y

  end function add_func

  subroutine printp(this)
    class(point), intent(in) :: this
    real :: tmp
    tmp = this%distance()  ! Call the type-bound function
    print *, 'Point: (x,y) = ', this%x, this%y, ' distance from orig = ', tmp
  end subroutine printp
end module class_point

program test
  use class_point
  implicit none
  type(point) :: p1,p2,p3     ! Declare a variable of type point.
  p1 = point(3.0,4.0)   !Use the implicit constructor
  call p1%printp         !Call the type-bound subroutine
  p2 = point(1.0,0.5)   ! Use the implicit constructor
  call p2%printp
```

```
  call p1%add(p2,p3)
  call p3%printp
print *, p1%add_func(p2)
end program test
```

Three things need to be noted. First, two `contains` appear here, one of which bounds the procedures names to the derived type and another is the same as the above, i.e., contains the definition of the procedures. Second, in the definition of the procedure, the first argument must be of the same type of the derived type and the declaration using the `class` keyword, instead of the `type` keyword. This a fundamental requirement for type-bound procedures. If declared using `type`, gfortran compiler would raise the following error: Non-polymorphic passed-object dummy argument. This is required by compilers to consider this procedure as a type-bound procedure rather than an ordinary one. Third, when calling the type-bound procedure, the object is passed to the procedure in the object-oriented way, e.g., `p1%print_point` rather than `print_point(p1)`.

Viktor K. Decyk's paper on this.

The code which modifies a given data structure should be localized or confined to one location in the program and not spread, uncontrollably, throughout the program. This property is called an encapsulation. In a sense, it is a generalization of the familiar notion of a function or subroutine.

http://www.cs.rpi.edu/~szymansk/OOF90/F90_Objects.html

Appendix B. Fortran best practice

It is a good practice to place procedures in modules (instead of internal procedures or external procedures) as it helps the compiler to detect programming errors and to optimize the code.

Subroutines should be kept reasonably short (e.g. about 50 lines (usually difficult) so that they can fill in one screen and thus are easy to read).

We now usually do not need to worry about the start up overheads involved in calling a subroutine, because some short subroutines can be "inlined" by compilers so that the inefficiency is generally not a problem.

Each `module` should be in a separate file.

Any code that introduces new physics to a code should have a switch to enable it to be turned off. This makes it possible to run the model in a configuration that is identical to the model before the new physics went in, in order to check that nothing unexpected has been broken.

Code should be accompanied by technical documentation describing the physical processes that the additional code is intended to model and how this is achieved. Any equations used should be documented (in their continuous form where appropriate) along with the methods used to discretise these equations.

Always bear in mind that somebody will have to maintain your code in the future. That person could be you several years later. Commenting the source lines that are not obvious from common senses. But do not make too extensive/detailed/specific comments because the details have good chance to become out-dated when codes are updated, which will make the comments inconsistent with the codes, causing confusion rather than clarification to code maintainers. Also, note that it is the code itself rather than the comment that is the final and absolute definition of what is exactly done in the code (i.e., every source code is self-documenting if you understand it). Therefore rather than investing much time in writing comments, we should spend more time in making the algorithm and code easy to understand even if there is no comment. Comments should be used to state the **reason** for doing something, instead of repeating the Fortran logic in words.

The goal of all these practice is to reduce the chance of introducing bugs.

All routines and documentation must be written using SI units. Standard SI prefixes may be used. Where relevant, the units used must be clearly stated in both code and documentation.

Try to find some ways to test your code. Do not ignore compiler warnings, as they may point you to potential problems.

http://jules-lsm.github.io/coding_standards/guidelines/best_practices.html

We tend to criticize other's coding style when we see a messy code. But think twice before criticizing.

At first glance of a code, your judgment of its readability is often false because it is based on the simple appearance of the code, i.e., whether the code is neat enough rather than whether its logical is clear/straightforward or not, ==>psychological reasons.

But we should realize that work and life seem to be always in a hurry.

http://annefou.github.io/Fortran/modules/modules.html

Ref:

https://www.phy.ornl.gov/csep/pl/node1.html (this is very good)

Appendix C. Stack vs Heap Memory allocation

Where are the stack and heap stored? They are both stored in the computer's RAM (Random Access Memory). Using RAM is faster than using virtual memory (paging file/swap file).

Stack frame access is easier than the heap frame as the stack allocation happens on contiguous blocks of memory and hence is cache friendly, but in case of heap frames which are dispersed throughout the memory so it cause more cache misses. Note that the above being continugous or dispersed in memory refers to the storage of different variables, For a single array, all the data element are gauranteed to be continous in memory regardless they are on stack or heap.

In a stack, the allocation and deallocation is automatically done, whereas, in heap, it needs to be done by the programmer manually.

However, it is generally better to consider "**scope**" and "**lifetime**" rather than "stack" and "heap".

Scope refers to what parts of the code can access a variable. Generally we think of **local scope** (can only be accessed by the current function) versus **global scope** (can be accessed anywhere).

Lifetime refers to when a variable is allocated and deallocated during program execution. Usually we think of **static allocation** (variable will persist through the entire duration of the program, making it useful for storing the same information across several function calls) versus **automatic allocation** (variable only persists during a single call to a function, making it useful for storing information that is only used during your function and can be discarded once you are done) versus **dynamic allocation** (variables whose duration is defined at runtime, instead of compile time like static or automatic).

Although most compilers and interpreters implement this behavior similarly in terms of using stacks, heaps, etc, a compiler may sometimes break these conventions if it wants as long as behavior is correct. For instance, due to optimization a local variable may only exist in a register or be removed entirely, even though most local variables exist in the stack. You are free to implement a compiler that doesn't even use a stack or a heap, but instead some other storage mechanisms (rarely done, since stacks and heaps are great for this).

A particularly poignant example of why it's important to distinguish between lifetime and scope is that a variable can have local scope but static lifetime. In the context of lifetime, "static" *always* means the variable is allocated at program start and deallocated when program exits.

The value of a local scalar can be retained between different calls to the `subroutine/function` (by using `save` keyword or compiler option to save all local variables) but automatic arrays can not have the `save` attribute.