Programming in Modern Fortran

Salvatore Filippone

School of Aerospace, Transport and Manufacturing,
salvatore.filippone@cranfield.ac.uk

IT4I, Ostrava, April 2016
The evolution of the Fortran language;
Modern Fortran structure: compilation units, modules, derived data types;
Object oriented design; Fortran object-oriented features
CoArrays: Fortran as a PGAS language
The PSBLAS and MLD2P4 software libraries: parallel sparse linear algebra;
Design patterns: using GPUs and CUDA code from Fortran
FORTRAN: originally FORmula TRANslation.

- Invented in the 50s by da John Backus, IBM;
- First “high level” language;
- First compiler ever;
- Multiple versions;
- First programming language standard: ANSI FORTRAN 66 (1966/72)

Still accounting for (at least) 50% of supercomputing applications (see PRACE Tier-0 application requirements and Benchmarks)
Recent history of the Fortran standard

- FORTRAN 77: published in 1978;
- Fortran 90: published in 1991; (originally Fortran 8X)
- Fortran 95: published in 1997;
- TR15581;
- Fortran 2003: published in 2004;
- Fortran 2008: published in 2010;
- Fortran 2015: currently in advanced approval stage;
The evolution of the Fortran language

Major changes:

**Fortran 90:** Free form, Recursion, Dynamic memory management, Pointers, Modules, Data types, Modules;

**Fortran 95:** Minor update, mainly fixing mistakes (TR15581).

**Fortran 2003:** Object-oriented programming via type extension; parametrized data types, C interoperability, IEEE arithmetic, ASYNCHRONOUS keyword;

**Fortran 2008:** Submodules, Coarrays, BLOCK construct
Standard jargon

Definition

**Language standard:** A contract between the Standards Committee, the vendor (compiler supplier), and the programmer (end user)

- Defines how to write a “legal” program;
- Constraints the actions of a compiler against a “legal” program;
- Does *not* constraint what happens with and “illegal” program;
- Leaves some semantics details open;
- Defines the language evolution through “features”:
  - **Obsolescent:** old ways for which there exist modern replacements;
  - **Deleted:** Things that are no longer legal
- What is declared “obsolescent” in a standard is expected to become “deleted” in the next
program hello

    write(*,*) "Hello world"

end program hello
A valid Fortran source code contains:

- Characters from the basic ASCII alphabet (no ISO/UNICODE accented letters outside of comments and strings);
- Decimal digits;
- Underscore `_`;
- Special characters: `+=-*/(),.$’"%&;<>?!&`
- Example:

\[
\begin{align*}
x &= y + & \\
& & \& \text{z} & ! \text{This is a comment}
\end{align*}
\]
Names

An entity name is built as follows:

- It starts with an alphabetic character;
- Contains alphanumeric characters and/or _
- Its (minimum) maximum length is 31 characters (i.e. all compilers must recognize at least 31 chars, but maybe more);
- Upper and lowercase letters are indistinguishable (outside of strings);

Some words may take a special meaning from the context, but there is no such thing as a reserved word, e.g. INTEGER, PROGRAM, DO
Names

```fortran
program program

  integer :: do, integer

  do integer=1,10
    do = integer
    print *, do, integer
  end do

end program program
```
Data Types

What is a data type?

A set of values;
The set of meaningful operations on the values;
The representation inside the program (system/processor)

They can be

Intrinsic:
  Numeric: Integer, Real, Complex
  Logical
  Character

Derived: A collection of objects having a type.
Data Types

What is a data type?

- A set of values;
- The set of meaningful operations on the values;
- The representation inside the program (system/processor)
What is a data type?

- A set of values;
- The set of meaningful operations on the values;
- The representation inside the program (system/processor)

They can be:

- **Intrinsic:**
  - Numeric: Integer, Real, Complex
  - Logical
  - Character

*Data Types*
What is a data type?

- A set of values;
- The set of meaningful operations on the values;
- The representation inside the program (system/processor)

They can be

- **Intrinsic:**
  - Numeric: Integer, Real, Complex
  - Logical
  - Character

- **Derived:** A collection of objects having a type.
Variable Declaration

- **Basic format:**

  ```fortran
  integer :: i
  real(kind(1.d0)) :: x
  complex(kind=dp) :: z
  character(len=20) :: filename
  ```

- **Attributes:**

  ```fortran
  integer, parameter :: nmax=200
  real(kind=dp), save, dimension(:,:) :: x
  real(kind=sp) :: y=1.0_sp
  character(len=*) :: name='Fortran course'
  ```
Numeric Types

- INTEGER
- REAL
- COMPLEX

together with the usual arithmetic operations.
What is exactly a Fortran INTEGER?
intrinsic: numeric enquiry

What is exactly a Fortran INTEGER? Model:

\[ i = s \times \sum_{k=1}^{q} w_k \times r^{k-1} \]
What is exactly a Fortran INTEGER? Model:

\[ i = s \times \sum_{k=1}^{q} w_k \times r^{k-1} \]

Analogously for reals:

\[ x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k} \]

together with the zero.

Note that IEEE 754 defines denormals and \(-0\).
**digits(x)**  Number of digits, i.e. $p$ or $q$;
**epsilon(x)**  Smallest real increment, $b^{1-p}$
**huge(x), tiny(x)**  Largest and smallest numbers, $r^{q-1}$ or $(1 - b^{-p})b^{e_{\text{max}}}, b^{e_{\text{min}}-1}$
**radix(x)**  $r$ or $b$
**maxexponent(x)**  $e_{\text{max}}$
**minexponent(x)**  $e_{\text{min}}$
**precision(x)**  Number of equivalent decimal digits
Data types can be chosen at compile time by specifying the range:

```
integer, parameter :: i6 = selected_int_kind(6)
integer(kind=i6) :: i, j, k
```

This is an integer going (at least) from -999999 to 999999

What is double precision?

```
integer, parameter :: sp = SELECTED_REAL_KIND(6,37)
integer, parameter :: dp = SELECTED_REAL_KIND(15,307)
real(kind=dp) :: doublevar
```

This abstracts the precision definition
Numeric types

Relational operators: map pairs of numbers to logicals

- `==` .eq.
- `=/=` .ne.
- `<` .lt.
- `<=` .le.
- `>=` .ge.
- `>` .gt.
Logical data

The usual values `.true.` and `.false.`

Operators:

- `.and.`
- `.or.`
- `.not.`
- `.eqv.`
- `.neqv.`
STRING operators:

- Substring: \texttt{name}(i:j)
- Concatenation: \texttt{name} // \texttt{surname}
- \texttt{lge(s1,s2)} less than or equ\L to operator for ASCII strings;
- \texttt{lgt lte llt} as above;
- \texttt{len(string)} length (\texttt{len\_trim(string)})
- \texttt{adjustl(string)}, \texttt{adjustr(string)}
- \texttt{trim(string)}
If Statement:

```fortran
if (a < b) c = 1

swap: if (a < b) then
    temp = a
    a = b; b = temp
else
    c = 1
end if swap
```

Note: the name `swap:` is optional.
Operators are not short circuited (as in C)

```fortran
if ((i<=n).and.(v(i)>0)) then  ! Segfault possible!!
```

The compiler may decide on the value of an expression based on a part of it. But it does not have to (as instead happens in C). Therefore

```fortran
if (i<=n) then
  if (v(i)>0) then  ! OK
```

Multiple selections, with no overlap:

```fortran
[name:] select case (i)
case(:-11)
    <block>
case(-10,-8,-6,-5:5,6,8,10)
    <block>
case(11:)
    <block>
case default
    <block>
end select [name]
```

Selectors must be a scalar INTEGER, LOGICAL, CHARACTER. The same can be achieved with ELSEIF, which is more flexible.
Iterations

\[
\text{do } i = 1, n \\
\quad a(i) = (b \times v(i)) + k \\
\text{end do}
\]

The loop variable \emph{must} be an integer

Most general \emph{bounded do}

\[
[\text{name:}] \text{ do } \langle \text{var} \rangle = \langle \text{exp1} \rangle, \langle \text{exp2} \rangle, \langle \text{exp3} \rangle \\
\quad \langle \text{block} \rangle \\
\text{end do } [\text{name}]
\]

The number of iterations is computed \emph{before} entering the cycle, as:

\[
\max((\exp2 - \exp1 + \exp3)/\exp3, 0)
\]
Unbounded DO

[name:] do
  <block>
end do [name]

How do you stop this iteration?
Unbounded DO

[name:] do
  <block>
end do [name]

How do you stop this iteration?
With the EXIT statement!!
DO WHILE expr
  do
    if (.not. <expr>) exit
    < block >
  end do
**Unbounded DO**

[name:] do  
[block]  
end do [name]  

How do you stop this iteration?  
With the EXIT statement!!

DO WHILE expr  
do  
[.not. <expr>] exit  
[block]  
end do

**REPEAT UNTIL expr**

do  
[block]  
if (<expr>) exit  
end do

Mixed form  

do  
[block1]  
if (<expr>) exit  
[block2]  
end do
itx = 0
restart: Do
   If (itx >= itmax) Exit restart
   < setup iteration >
iteration: Do
   itx = itx + 1
   < iteration body 1 >
   If (sigma==zero) Then
      ! Breakdown, try restarting
      Exit iteration
   Endif
   < iteration body 2 >
   If (rerr<=eps) Then
      ! Reached convergence
      Exit restart
   End If
End Do iteration
End Do restart
Iterations: DO

CYCLE

  do
  < block1 >
  if (<expr>) cycle [name]
  < block2 >
  end do

goes to the next iteration of cycle [name]
subroutine foo(a,b,c)
    integer, intent(in) :: a,b
    integer, intent(out) :: c
    c = a + b
end subroutine foo

program silly
    integer x, y, z
    x=1
    y=2
    call foo(x,y,z)
    write(*,*) z, bar(z)
contains
    function bar(a)
        integer, intent(in) :: a
        integer :: bar
        bar = 2*a
    end function bar
end program silly
Array language

Array: collection of items all of the same type, with a RANK and a SHAPE:

```fortran
real(kind=dp), dimension(50,50) :: a
real(kind=dp), dimension(1:20,0:30) :: b
```

Storage “by columns”: leftmost index varies most rapidly
You can specify a subset of an array with a *triplet* (start:end:stride):

```fortran
a(1:20,2:30)
b(10:20,1:20:2)
a(3,:) 
b(i,10:1:-1)
```
program tryshape
    use typekinds
    
    real(kind=dp), dimension(50,50) :: a
    real(kind=dp), dimension(1:20,0:30) :: b
    integer :: i=3
    
    b = b * a
    print *,shape(a(1:20,2:30)) !
    print *,shape(b(10:20,1:20:2)) !
    print *,shape(a(3:3,:)) !
    print *,shape(b(i,10:1:-1)) !
end program tryshape
Array language

Operators work on array expressions:

\[
\text{real(kind=dp)}, \ \text{dimension}(20,20) :: a, b, c
\]

\[
\text{call compute(a)}
\]

\[
b = a
\]
\[
c = 2*a - b
\]
\[
a = c / b
\]

All operators are intended element by element
Array language

All arrays in a given expression must have the same SHAPE (conformant):

```
real(kind=dp), dimension(50,50) :: a
real(kind=dp), dimension(1:20,0:30) :: b
real(kind=dp), dimension(20,20) :: c
real(kind=dp) :: d
```

c = a * b

! Illegal

c = a(1:20,31:50) + b(:,0:19)

! OK

c(2,:) = 2.0*a(21:40, 45) + d

! Scalar conforms

! to any shape
Array language

Example: triangular system solve classic:

```fortran
  do i=1,n
    x(i) = x(i) / a(i,i)
    do j=i+1, n
      x(j) = x(j) - x(i) * a(j,i)
    end do
  end do
```

With the array language:

```fortran
  do i=1,n
    x(i) = x(i) / a(i,i)
    x(i+1:n) = x(i+1:n) - x(i) * a(i+1:n,i)
  end do
```

The value of an expression is computed \textit{before} assignment:
\[ \mathbf{v}(2:10) = \mathbf{v}(1:9) \]

Basic rule: \textit{the order of execution of the individual arithmetic instructions must be irrelevant} (hence the compiler is free to reorder loops for performance).
Sometimes this entails temporary copies.
Array language

It is possible to specify constant arrays

\[ v(:) = (/ 1, 2, 3, 4 /) \]

! F2003

\[ v(:) = [ 1,2,3,4 ] \]

And array indexing

\[ A( (/1, 2, 3, 4, 5 /) ) = XX(\ldots) \]

Any repeated value in an index array is only legal on the right hand side of an assignment.
Array intrinsics

`lbound(a [,dim])` Array of lower bounds
`ubound(a [,dim])` Array of upper bounds
`size(a [,dim])` Size (total or per dimension)
`shape(a)` Shape of the array
`matmul(a,b)` Matrix multiplication

Many compiler have a switch to link an external BLAS-like function for this, in GNU fortran: `-fexternal-blas`
Array intrinsics

Logical reductions:
- all( <expr> )
- any( <expr> )
- count( <expr> )

Numeric reductions
- maxval (array), minval (array)
- maxloc (array [,dim,mask]), minloc (array [,dim,mask])
- product (array), sum (array)

Note: floating-point reductions are affected by rounding.
Conditionals in array expressions: WHERE statement

```fortran
where ( a > 0.0 ) a = 1.0/a
```

WHERE construct

```fortran
where ( a > 0.0 )
  b = 1.0 / a
  a = 2.0 * a
elsewhere
  a = 2.0
  b = 1.0
end where
```
First kind: automatic objects

```fortran
subroutine swap(a,b)
  real, dimension(:) :: a,b
  real, dimension(size(a)) :: work

  work = a
  a = b
  b = work
end subroutine swap
```

May or may not be allocated on the stack/heap, depending on the compiler.
program tryall
   real, dimension(:), allocatable :: v
   integer :: n

   print *, 'Size ? '
   read *, n

   allocate(v(n))
   call do_something(v)

   deallocate(v)
end program tryall

Argument of an ALLOCATE must be an ALLOCATABLE or a POINTER.
subroutine do_something(v)
  real, dimension(:) :: v
  real, allocatable :: temp(:)
  integer :: n, info

  n = size(v)

  ! Checking for successful allocation
  ! highly recommended
  allocate(temp(0:n-1), stat=info)
  if (info /= 0) then
    print *, 'Memory allocation failure'
  end if

  if (allocated(temp)) then
    ......
  end if

  deallocate(temp, stat=info)
  if (info /= 0) print *, 'Deallocation failure?'
end subroutine do_something
Dinamic Memory: ALLOCATE

A variable may be SAVEd:

```
subroutine do_something(n,v)
    integer :: n
    real, dimension(:) :: v
    real, allocatable, save :: temp(:)

    if (allocated(temp)) then
        if (size(temp) < n) deallocate(temp)
    end if
    if (.not.allocated(temp)) then
        allocate(temp(0:n-1),stat=info)
        if (info /= 0) then
            print *, 'Memory allocation failure'
        end if
    end if
end subroutine do_something
```
**ALLOCATABLE Semantics**

1. **SAVE** refers to both the allocation status and the contents of the variable.
2. If a variable is not **SAVE**d, it is automatically deallocated when it goes out of scope.
3. It is an error to **ALLOCATE** and allocated variable or **DEALLOCATE** a non-allocated variable.
4. There is no explicit **REALLOCATE**, but there exists **MOVE_ALLO**C and allocation on assignment.
5. An array must be **ALLOCATED** before use (except for the LHS of an assignment).

What is the difference with a **POINTER**?

- An **ALLOCATABLE** is always uniquely defined;
- An **ALLOCATABLE** always has a well defined state.
The typical structure of a PROGRAM is

```fortran
[program <program-name>]
    [<specifications>]
    [<executable stmts>]
[contains]
    <internal-subprograms>]
end [program <program-name>]
```

If you have subroutine instead of program, you obtain an external subprogram, that is, a “classic” subroutine
Internal subprograms

- May appear inside PROGRAM or SUBROUTINES
- Have access to the variables of the container unit;
- They are typically convenient for small pieces of codes;
- Their interface is automatically available
We have seen an example of an **ASSUMED SHAPE ARRAY**:

```fortran
subroutine swap(a,b)
real, dimension(:) :: a,b
```

The compiler must pass around some information; this is possible *IF* it knows that it must be done, but then, the *interface* must be known.

**Definition**

An interface is the description of names, types and attributes of dummy arguments

By default we get an *implicit interface*, which in this case gives an **ASSUMED SIZE ARRAY**

```fortran
subroutine swap(a,b)
real :: a(*),b(*)
```

of which we know only the starting address. intsub1.f90
Explicit interfaces may be provided by using internal subroutines, the INTERFACE statement, or MODULES.
With the INTERFACE statement

```fortran
program intsub
  implicit none
  interface
    subroutine swap(a,b)
      real, dimension(:), intent(inout) :: a,b
    end subroutine swap
  end interface
end program intsub
```
Argument passing

Dummy Arguments have the following attributes:

- **INTENT**: IN, OUT, INOUT how it can be used inside the subroutine;
- **OPTIONAL** May be omitted from the actual call;
- **TARGET** can be used with a POINTER
Argument passing

```fortran
interface
  subroutine itsol(a,b,x,itmax,tol,stopc,iter,err)
      real(kind=dp), intent(in) :: a(:,,:), b(:) :: a(:,:), b(:)
      real(kind=dp), intent(inout) :: x(:) :: x(:)
      integer, optional, intent(in) :: itmax, stopc :: itmax, stopc
      real(kind=dp), optional, intent(in) :: tol :: tol
      real(kind=dp), optional, intent(out) :: err :: err
      integer, optional, intent(out) :: iter :: iter
  end subroutine itsol
end interface
```
Argument passing

You can call by keyword

```fortran
    call itsol(a,b,x)
call itsol(a,b,x,200)
call itsol(a,b,x,itmax=200,err=rout,tol=1.e-5_dp,iter=it)
```

In the implementation:

```fortran
    if (present(itmax)) then
        itmax_ = itmax
    else
        itmax_ = 100
    endif
<....>
iteration: do i=1, itm
<....>
    if (err_ < tol) exit iteration
<....>
end do iteration
if (present(err))   err = err_
```
Last type of compilation unit is the MODULE

```
module  <module-name>]
    [<specifications>]
[contains
    <module-subprograms>]
end [module  <module-name>]
```

What is it used for?
First use: group together functions and make available their interfaces

module vecutil
contains
  subroutine swap(a,b)
    real, dimension(:) :: a,b
    real, dimension(size(a)) :: work
    work = a
    a = b
    b = work
  end subroutine swap
end module vecutil

program tryswap
use vecutil

module vecutil
  use typekinds
  interface swap
    module procedure spswap, dswap, iswap
  end interface
contains
subroutine spswap(a,b)
  real(kind=sp), dimension(:) :: a,b
  real(kind=sp), dimension(size(a)) :: work

  work = a
  a = b
  b = work
end subroutine spswap
Generic interfaces: overloading

```fortran
subroutine dpswap(a,b)
    real(kind=dp), dimension(:) :: a,b
    real(kind=dp), dimension(size(a)) :: work
    work = a
    a = b
    b = work
end subroutine dpswap

subroutine iswap(a,b)
    integer, dimension(:) :: a,b
    integer, dimension(size(a)) :: work
    work = a
    a = b
    b = work
end subroutine iswap

end module vecutil
```

tryswp.f90
Second use: define shared data

```fortran
module data_area
  integer, dimension(:), allocatable :: intvec
  real, dimension(:), allocatable :: realvec
end module data_area
```

```fortran
program try_data
use data_area
allocate(int_vec(n),realvec(n))
.....
```
### Summary: Program Structure

| :---: | :---: | :---: |
| **PROGRAM, SUBROUTINE, FUNCTION, MODULE, BLOCK DATA** | **USE** | **IMPLICIT NONE** |
| **FORMAT, ENTRY** | **PARAMETER** | **IMPLICIT** |
| | **PARAMETER, DATA** | **Type definition, interface, type declarations** |
| | **DATA** | **Executable statements** |
| **CONTAINS** | **Internal or module procedures** |
| **END** |
What happens if we have to deal with objects having a complicated internal structure?
Derived Data Types

What happens if we have to deal with objects having a complicated internal structure?
Example: sparse matrices: type, row indices, column indices, auxiliary info, etc

Cannot use *independent* arguments:
- Code becomes unreadable (way too many arguments);
- Implementation is hardwired (portability and maintainability issues);
Derived Data Types

Group together items with (potentially) different types

type point2d
  real :: x, y
end type point2d

function distance2d(p1, p2)
  real :: distance2d
  type(point2d), intent(in) :: p1, p2
  real :: tempx, tempy

  tempx = abs(p2%x - p1%x)
  tempy = abs(p2%y - p1%y)
  distance2d = sqrt(tempx**2 + tempy**2)
end function distance2d

testgeo.f90
module geometry
  interface distance
    module procedure distance2d, distance3d
  end interface

............

We can therefore define both objects and operators. "specific” procedures must be distinguishable
Our objective is to write code like

program geotest
use geometry

type(point2d) :: a, b, c

a=point2d(1.0,2.0)
b=point2d(3.0,4.0)
c = a + b
Generic interfaces: operator overloading

```
interface  [operator(<operator name>) | assignment(=)]
    [interface body]
end interface
```

- operator is arbitrary; you can also invent new operator names between dots: `.squash.`
- All operator procedures must be function with one or two arguments `intent(in)`
- For assignment, you have to create a subroutine with one `intent(in)` argument and one `out` or `inout`
- It is forbidden to redefine default operators on intrinsic types.

Example: the MPF90 multiprecision package.
USE statement

To access a module you have to USE it. However this might create name conflicts; moreover, it is good practice to make clear what are you using from a module, and where entities come from

use vecutil, only : swap => iswap

It is moreover good practice to use PUBLIC and PRIVATE

module util
    public get_number
    integer, private :: i
contains
    integer function get_number
        get_number = i
    end function get_number
end module util

tryscope.f90
In a variable declaration we can have the `pointer` attribute:

```fortran
integer, pointer :: pi
real, pointer :: pr
```

It is needed to:
- Implement data structures with arbitrary linkage (linked lists and trees);
- To have “views” of existing arrays;

A pointer can be allocated

```fortran
allocate(pi)
pI = 1
print *, pi
```
A POINTER can be used as a “view” of an array:

```fortran
real, target, allocatable :: a(:,::)
real, pointer :: pa(:,::)
```

```fortran
allocate(a(-n:n,-n:n))
```

```fortran
do i=-n, n, 2 ! assuming n is even
   pa => a(i:i+1,i:i+1)
   call do_something_2x2(pa)
end do
```

Status check

```fortran
ASSOCIATED(PNT [, TRGT])
```

To cancel an association:

```fortran
NULLIFY(p)
p => null()
```
Suppose you want to implement a list-based structure such as a stack, i.e. an object defined with two operations:

1. **PUSH(ITEM,STACK)**: takes an object and puts it on top of the stack;
2. **POP(ITEM,STACK)**: Removes and returns the first object on the stack.
Basic definitions:

```fortran
type basetype
  integer :: data
end type basetype

type stackitem
  type(basetype) :: data
  type(stackitem), pointer :: next=>null()
end type stackitem
```

Now, implement a STACK
POINTERs tips and tricks:

- It is always possible to assign an allocated POINTER to another POINTER;
- An object allocated through a POINTER implicitly gets the TARGET attribute;
- No garbage collection;
- No array of POINTERS (but you can have an array of a TYPE containing a pointer);
- A pointer may point to a procedure.

More in the OOP section.

Should you use POINTERs?

*Never use a POINTER when an ALLOCATABLE suffices*
How to handle the compilation process

The Fortran standard does not specify how MODULEs should be implemented. The most common scheme is:

- For each module `<name>` statement the compiler emits a file `name.mod`
- Any time a `use <name>` is encountered, the file `name.mod` must be accessed (typically with the `-I` option)
- If (as is often the case) the file `name.f90` contains the source code for module `name`, then at link time you must include `name.o`
- Any time the file `name.mod`, you should recompile all sources where you have `use name`
How to handle the compilation process

1. Compile the module file:
   - `module_file.f90`
   - Use `f90_compiler`
   - Output: `module.mod` and `module_file.o`

2. Compile the program file:
   - `program.f90`
   - Use `f90_compiler`
   - Output: `program.o`

3. Link the objects:
   - `module.mod`
   - `module_file.o`
   - Use `linker`
   - Output: `program`

The process is represented in a diagram.
How to handle the compilation process

This is a heavy process: we should only recompile when interfaces change.

How to separate interface from implementation? Solution 1: Manually implement the functionality in an external procedure

```fortran
module foo_type
    type bar
    end type bar
end module
module foo_mod
    interface
        subroutine foobar(x)
            use foo_type
            type(bar), intent(inout) :: x
        end subroutine
    end interface
end module

subroutine foobar(x)
    use foo_type
    type(bar), intent(inout) :: x
end subroutine
```

This:

- Requires the programmer to maintain consistency by hand since you can not include foo_mod in the implementation of foobar.
- Prevents the use of PRIVATE entities, since they would be invisible outside foo_mod.
How to handle compilation: submodules

The correct solution is provided by Fortran 2008 SUBMODULEs

```fortran
module foo_mod
  type bar
  end type bar
  interface
    module subroutine foobar(x)
      import :: bar
      type(bar), intent(inout) :: x
    end subroutine
  end interface
end module

submodule (foo_mod) foo_mod_impl
contains
  module subroutine foobar(x)
    type(bar), intent(inout) :: x
    ...
  end subroutine
end subroutine
end submodule
```

Note:
- IMPORT allows to use the type bar in the interface;
- If you spell out the arguments to foobar in the submodule, the compiler will check consistency for you.
The BLOCK construct

Allows declaration of local variables

do  i=1,m
  block
    real  alpha, temp(m)
    temp(j) = alpha*a(j,i) + b(j)
  end block
end do
The \textbf{BLOCK} construct

Allows declaration of local variables

\begin{verbatim}
  do  i=1,m
    block
      real  alpha, temp(m)
      temp(j) = alpha*a(j,i) + b(j)
    end block
  end do
\end{verbatim}
The ASSOCIATE construct

Allows creating a local alias for a name

```fortran
associate(point_state => &
    & master_list%item(n)%qfield%posn(i,j)%state)
point_state%xv = get_posn_vec(x,z)
point_state%t  = timestep(t)
end associate
```
Resources

- Metcalf & Reid & Cohen: Modern Fortran explained, Oxford
- Arjen Markus: Modern Fortran in Practice, Cambridge
- http://www.dmoz.org/Computers/Programming/Languages/Fortran/Tools/
- http://www.personal.psu.edu/faculty/h/d/hdk/fortran.html
- http://www.fortran.com/metcalf.htm
- comp.lang.fortran
- gcc.gnu.org