Modules

Package together into one file declarations of variables and procedures related to the same task.

```
module m
! type and variable declarations
...
contains
subroutine sub
!

end
...
end module
```

Procedures in a module can have its own internal procedures.

A module and declarations inside the module can be used with the `use`-statement,

No `common` areas are needed, since all decklarations inside a module are automatically visible to the program using the module.

A module is compiled separately and linked to user's program.

Example: definition of constants.

```
module realnum
 implicit none
 integer, parameter :: single=selected_real_kind(5)
 integer, parameter :: double=selected_real_kind(10)
end module
```

usage:

```
program realtest
  use realnum
  implicit none
  real (kind=single) :: x
  real (kind=double) :: y
  ...
end program
```

The module is compiled (with option `-c`) and the object code is stored into a file `real-num.o`. The file is not executable.

The module is then linked to a program using it:

```
f95 -c realnum.f90
f95 -o realtest realnum.o realtest.f90
```

Use statement

Must be before all other declarations.

Obejcts inside a module can be referenced also by other names:

```
 program realtest
   use realnum, myreal=>double
   implicit none
   real (kind=myreal) :: x
   ...
end program
```

To use single precision, just change `myreal=>double` to `myreal=>single`. No other changes are needed.

The default is that all declarations in a module will be visible. This can also be restricted:

```
program realtest
  use realnum, only: double
   ...
end program
```

tai

```
program realtest
  use realnum, only: myreal=>double
   ...
end program
```

- Explicitly seen what declarations are used.
- Avoid conflicts between own variables and module variables with the same name.

Example: a module for handling rational numbers.

We would like to write a program like

```
program koe
  use ratpack
  implicit none
  type (rational) :: p, q, r
  p=rat(1,2)
  q=rat(2,3)
  r=p+q
  q=q+1
  write(*,*) nomin(r),denom(r)
  write(*,*) nomin(q),denom(q)
  p=5
  p= .inv. p
  write(*,*) nomin(p),denom(p)
end program
```

Output of the program:

```
7 6
5 3
1 5
```

A simple version contains the definition of the type rational and functions for basic arithmetic operations.

```
module ratpack
type rational
  integer :: nominator, denominator
end type
contains

function mul (p, q)
! product of rational numbers p and q
 type (rational) mul
 type (rational), intent(in):: p, q
 integer a,b,c,d
 type (rational) r
 a=p%nominator; b=p%denominator
 c=q%nominator; d=q%denominator
 r%nominator = a*c
 r%denominator = b*d
 mul = simplify (r)
 return
end function
```

```
function div (p, q)
! quotient of rational numbers p and q
..
end function

function add (p, q)
! sum of rational numbers p and q
..
end function

function sub (p, q)
! difference of rational numbers p and q
..
end function
```

After each operation rational number must be reduced to the simplest form to prevent the nominator and denominator from growing ever bigger.

```
function simplify (q)
! reduce the rational number q to its simplest form
! using the Euclidean algorithm
! the value of the function is the reduced number
type (rational) simplify
 type (rational), intent(in):: q
 integer  a, b
 a=q%nominator
 b=q%denominator
 do
  if (a < b) then
    c=a; a=b; b=c;
  end if
  do while (a >= b)
   a = a-b
  end do
  if (a == 0) exit
 end do
```

```
  ! b is now the greatest common factor
  ! of nominator and denominator
  simplify%nominator = q%nominator/b
  simplify%denominator = q%denominator/b
  return
end function

end module
```

Using the package:

```
   program koe
     use ratpack
     implicit none
     type (rational) :: p, q, r

     p%nominator=1;  p%denominator=2
     q%nominator=3;  q%denominator=4
     r=add(p,q)
     write(*,*) r%nominator, r%denominator
   end program
```

```
 f95 -o koe ratpack.o koe.f90
```

After the `use` statement all types, variables and procedures declared in the module can be used. The declarations are public.

A module may also contain private declarations that the user cannot access.

```
real, private, dimension(100) :: table
```

Even components of a public data type can be private:

```
type rational
  integer, private :: nominator, denominator
end type
```

Now the user cannot directly access the components of the type `rational`. The components can be set and read only by using procedures declared in the module.

The internal representation of the type can be changed without affecting user's programs. Names of components or even the whole method of representation can be changed.

Safe for data structures. The user cannot mess them up accidentally.

If the internal representation of rational numbers is private, we'll need a procedure in the module (having access to the nominator and denominator) for setting values of rational numbers:

```
 function rat (n, d)
 ! return a  rational number with
 ! nominator=n denominator=d
  type (rational) rat
  integer, intent(in):: n, d
  rat%nominator=n
  rat%denominator=d
  return
end function
```

We'll also need procedures for finding the nominator and denominator of a rational number:

```
 function nomin (p)
 ! return the nominator of a rational number
  integer nomin
  type (rational), intent(in):: p
  nomin=p%nominator
  return
 end function

 function denom (p)
 ! return the denominator of a rational number
  integer denom
  type (rational), intent(in):: p
  denom=p%denominator
  return
end function
```

Now the nominator and denominator cannot be accessed directly. Instead, we have to use procedures of the module:

```
 program koe
   use ratpack
   implicit none
   type (rational) :: p, q, r

   p=rat(1, 2)
   p=rat(3, 4)
   r=add(p,q)
   write(*,*) nomin(p), denom(p)
end program
```

Generic procedures

The function `max` is an example of a generic procedure. Depending on the types of the actual arguments it can return an integer, a real number, or a double precision number.

Generic procedure is a common name for a set of different procedures, one of which is selected to be invoked depending on the types of the actual arguments.

Different versions of the procedure must have different types of formal parameters so that it is possible to decide which one will be executed. Just reordering the arguments is not enough, since keyword parameters can be used.

```
module div2
private
interface half
   module procedure half1, half2
end interface
public :: half

contains

integer function half1(i)
  integer, intent(in) :: i
  half1 = i/2
end function

real function half2(x)
  real, intent(in) :: x
  half2 = x/2.0
end function
end module
```

Procedure `half` refers to two different procedures, one of which will be actually executed. Here `half1` and `half2` are private, and cannot be called directly.

```
 program divtest
  use div2
  integer :: i=10
  real :: x=1.0
  i=half(i)
  x=half(x)
  write(*,*) i,x
end program
```

Operator overloading

In principle an operator is a function of one or two variables. It can also be a generic function. In the case of opreators the order of arguments is essential.

Continuing the example of rational numbers: Define a function for addition separately for all possible type combinations.

```
interface operator (+)
   module procedure ratint_add, intrat_add, add
end interface

...
contains
...

function add (p, q)
! sum of rational numbers p and q
 type (rational) :: add
 type (rational), intent(in):: p, q
  ...
  ! this is the previously defined function
  ...
end function
```

```
 function ratint_add (p, i)
 ! lsum of a rational number p and integer i
  type (rational) :: ratint_add
  type (rational), intent(in):: p
  integer, intent(in) :: i
  integer a,b,c,d
  type (rational) :: r
  a=p%nominator; b=p%denominator
  c=i; d=1
  r%nominator = a*d + b*c
  r%denominator = b*d
  ratint_add = simplify (r)
  return
end function
```

```
 function intrat_add (i, q)
 ! lsum of an integer i and a rational number q
  type (rational) :: intrat_add
  integer, intent(in) :: i
  type (rational), intent(in):: q
  integer a,b,c,d
  type (rational) :: r
  a=i; b=1
  c=q%nominator; d=q%denominator
  r%nominator = a*d + b*c
  r%denominator = b*d
  intrat_add = simplify (r)
  return
end function
```

Assignment

If both sides of an assignment statement are of the same type, no extra declarations are needed.

If the assignment requires a non standard type conversion, we have to define what the assignment operator will do.

For example, ”rational number = integer” is not automatically defined. We have to explain how the conversion should be made.

```
interface assignment (=)
  module procedure intrat_subst
end interface
...

subroutine intrat_subst (p, i)
! assignment rational number p = integer i
  integer, intent(in) :: i
  type (rational), intent(out) :: p
  p%nominator = i
  p%denominator = 1
  return
end subroutine
```

Another example: magnitude arithmetic. Magnitudes are logarithmioc quantities and cannot be added. If we want to know the total magnitude of a binary star, we have to first calculate the corresponding flux densities.

```
module magnitude
  implicit none
  type magnitude
     real::m
  end type
  interface assignment (=)
     module procedure magtoreal, realtomag
  end interface
  interface operator (+)
     module procedure addmag
  end interface
contains
```

Magnitude contains just a real number, but the value can be changed only using appropriate functions that will not do very much:

```
subroutine magtoreal(x, mag)
! assignment x=mag, convert  magnitude -> real
 real, intent(out) :: x
 type (magnitude), intent(in) :: mag
 x = mag%m
end subroutine

subroutine realtomag(mag, x)
! assignment mag=x, convet real->magnitude
 real, intent(in) :: x
 type (magnitude), intent(out) :: mag
 mag%m  = x
end subroutine
```

Addition of magnitudes: first convert to flux densities, add and convert back to magnitudes.

```
function addmag(m1, m2)
! "addition" of magnitudes
 type (magnitude) :: addmag
 type (magnitude), intent(in) :: m1, m2
 real f
 f = 10**(-0.4*m1%m) + 10**(-0.4*m2%m)
 addmag%m = -2.5*log10(f)
end function
end module
```

Finally a main program to test our module:

```
 program magtest
  use magnitude
  implicit none
  type (magnitude) :: m1, m2, mtot
  real x
  m1 = 1.0
  m2 = 2.0
  mtot = m1+m2
  x=mtot
  write(*,*) x
end program
```

```
0.6361488
```

Own operators

It is possible to define own operators.

Dotted notation only (kuten `.eq.`, `.and.` jne)

```
 interface operator (.inv.)
   module procedure inverse
 end interface
 ...

 function inverse (p)
 ! inverse of the rational number p
 type (rational) inverse
 type (rational), intent(in):: p
 type (rational) r
 r%denominator=p%nominator
 r%nominator=p%denominator
 inverse = simplify (r)
 return
end function
```

It is convenient to pack frequently needed constants, type definitions and procedures into modules.

Think which declarations should be kept private. It is not necessary that then user can access everything, since that may lead to errors that are difficult to trace. Especially when hadling complicated data structures it is better to provide the user just with a set of well tested and reliable tools.

Think carefully which operations of own data types should be defined as operators. Excessive use of operators may make the program obscure.

Arithmetic operations must be natural or intuitive (what could the product of two magnitudes mean?)