CLM Coding Conventions

>>Terms of Use
>>Go BACK to previous page.

This page is intended as a guide for CLM coding standards, practices and conventions. On this page, we focus on five items that we want all developers to concentrate on. Keep these in the back of your mind as you are programming and refer to them if you have a question. We are more than willing to work with you throughout the process (contact us at CLM-CMT@cgd.ucar.edu). If code doesn't generally conform to these items then it will delay the tagging process and we will ask you to modify your code accordingly.

Our past experiences have led us to document a more Comprehensive list of standards that we are striving for. You will find constructs that do not follow these guidelines and we are updating the legacy CLM code base as time and resources permit.

Comments

1. Comment often and include explanations for algorithms and processes. Your code should be easily readable and understandable by anyone else who reads your work.
    a. **Logical Branches** - include a comment after an **else** or **endif** statement. Use something that makes sense for the type of logical branch you are using. This is particularly important when there are large numbers of nested blocks. Look at the second if statement below for an even nicer way to accomplish the same thing without comments.

Comments and Logical Branches

```
if ( someLogical ) then ! someLogical controls the assignment of x or y
   x = 0.0_r8
else ! else someLogical
   y = 0.0_r8
endif ! if someLogical

! Alternative to comments
fates_if: if (use_fates) then
   lch4_if: if (use_lch4) then
[...]
   end if lch4_if
end if fates_if
```

    b. **Pointer Intent** - If you pass information to a subroutine via a pointer in an associate block, include a comment as to the **intent** (OUT, IN, INOUT) of that pointer. This is in addition to a description of that pointer as well as the type and array dimensions. Also see **Argument Passing** on this page.

Comments and Pointer Intent

```
associate(&
   ptrIn    => derivedType%readOnlyInRoutine      , & ! [character(len=8) (:)] intent = in    used
for x
   ptrInOut => derivedType%readAndWriteInRoutine  , & ! [real(r8) (:,:,:)]     intent = inout used
for y
   ptrOut   => derivedType%writeOnlyInRoutine       & ! [real(r8) (:)]         intent = out   used
for z
)
...
end associate
```

        i. The meaning (semantic) of **intent** (out, in, inout)
            1. out - a variable is only set within a subroutine
            2. in - a variable is only passed into and used by a subroutine
            3. inout - a variable may be both modified and used within a subroutine

Argument Passing

1. The **intent** attribute of subroutine input/output variables is used. Argument lists should ***always*** use the **intent** attribute (try to have data be either IN or OUT and avoid INOUT if possible/reasonable).
2. There are currently three ways in the current code with which you can pass information to subroutines. By interface (best choice), by pointer (second best choice) and by using a global data type (a bad option, but sometimes still necessary within the current CLM framework). We are in the process of refactoring the code so that all data will be passed through the subroutine interface. New code coming in should not pass via global data unless all other options do not work.
    a. **Subroutine Interface** - Use the intent keyword and keyword pairs if you have more than a few arguments. Try to consistently pass either all derived types or all FORTRAN types. Passing derived types can shorten the list of arguments and make it more understandable. Below are two examples from CLM, (subsection i, below) and RTM, (subsection ii, below), respectively.
        i. CLM example from BandDiagonal::BandDiagonalMod.F90
            1. Subroutine definition

Interfaces::CLM. Example from BandDiagonal, definition

```
subroutine BandDiagonal(bounds, lbj, ubj, jtop, jbot, numf, filter, nband, b, r, u)
    !
    ! !DESCRIPTION:
    ! Tridiagonal matrix solution
    !
    ! !ARGUMENTS:
    implicit none
    type(bounds_type), intent(in) :: bounds                    ! bounds
    integer , intent(in)    :: lbj, ubj                        ! lbinning and ubing
level indices
    integer , intent(in)    :: jtop( bounds%begc: )            ! top level for each
column [col]
    integer , intent(in)    :: jbot( bounds%begc: )            ! bottom level for
each column [col]
    integer , intent(in)    :: numf                            ! filter dimension
    integer , intent(in)    :: nband                           ! band width
    integer , intent(in)    :: filter(:)                       ! filter
    real(r8), intent(in)    :: b( bounds%begc: , 1:   , lbj: ) ! compact band matrix
[col, nband, j]
    real(r8), intent(in)    :: r( bounds%begc: , lbj: )        ! "r" rhs of linear
system [col, j]
    real(r8), intent(inout) :: u( bounds%begc: , lbj: )        ! solution [col, j]
    !
    ! ! LOCAL VARIABLES:
    integer  :: j,ci,fc,info,m,n             !indices
    integer  :: kl,ku                        !number of sub/super diagonals
    integer, allocatable :: ipiv(:)          !temporary
    real(r8),allocatable :: ab(:,:),temp(:,:) !compact storage array
    real(r8),allocatable :: result(:)

    !------------------------------------------------------------------------

    ! Enforce expected array sizes
    SHR_ASSERT_ALL((ubound(jtop) == (/bounds%endc/)),             errMsg(__FILE__,
__LINE__))
    SHR_ASSERT_ALL((ubound(jbot) == (/bounds%endc/)),             errMsg(__FILE__,
__LINE__))
    SHR_ASSERT_ALL((ubound(b)    == (/bounds%endc, nband, ubj/)), errMsg(__FILE__,
__LINE__))
    SHR_ASSERT_ALL((ubound(r)    == (/bounds%endc, ubj/)),        errMsg(__FILE__,
__LINE__))
    SHR_ASSERT_ALL((ubound(u)    == (/bounds%endc, ubj/)),        errMsg(__FILE__,
__LINE__))
```

            2. The definition above illustrates a few important points.
                - Specify the lower bounds of array arguments (this is needed for the subroutine to operate properly in a threaded region)
                - Do **not** specify the upper bounds of array arguments (doing so can prevent the compiler from checking for array size agreement between the caller and callee, and can carry a performance penalty)
                - The first set of code in the routine should be assertions for the expected array upper bounds
            3. The above code demonstrates a few stylistic points as well:
                - Specify the lower bound explicitly, even when it is 1 (see the 'b' argument, above)
                - Put a space after each ':' for readability
                - Add comments about expected array sizes, as in [col, nband, j]

4. Subroutine call
   Follow this example in which BandDiagonal is called from SoilTemperatureMod.F90:

Interfaces::CLM. Example from BandDiagonal, call

```
call BandDiagonal(bounds, -nlevsno, nlevgrnd, jtop(begc:endc), jbot(begc:endc), &
         num_nolakec, filter_nolakec, nband, bmatrix(begc:endc, :, :), &
         rvector(begc:endc, :), tvector(begc:endc, :))
```

   - The important point here is that the lower and upper bounds are explicitly specified for the gridcell / landunit / col / pft dimension of all array arguments. This is important for threading to work properly.
   - Even for local variables whose dimensions only go begc:endc, please explicitly specify the dimensions as above when passing the array to a subroutine. This will ensure that threading continues to work if the variable's dimensions change (e.g., if it is pulled into clmtype, and thus has dimensions that span the proc_bounds rather than the clump_bounds).

ii. RTM example from RtmMod.F90

Interfaces::RTM. Example from RtmMod.F90

```
!
! The call of the routine
!
call RtmFloodInit (frivinp_rtm, begr, endr, nt_rtm, &
     runoff%fthresh( begr:endr ),                   &
     evel( begr:endr , : ),                         &
     runoff%gindex( begr:endr ),                    &
     runoff%lnumr,                                  &
     flood_active,                                  &
     effvel_active)
...
!
! The interface definition
!
subroutine RtmFloodInit(frivinp, begr, endr, nt_rtm, fthresh, evel, &
     gindex ,        &
     lnumr ,         &
     is_rtmflood_on, &
     is_effvel_on    )
...
   ! Subroutine arguments
   ! in mode arguments
   character(len=*), intent(in) :: frivinp
   integer ,         intent(in) :: begr, endr, nt_rtm
   logical ,         intent(in) :: is_rtmflood_on      !control flooding
   logical ,         intent(in) :: is_effvel_on        !control eff. velocity
   integer ,         intent(in) :: gindex( begr: )     ! global index [begr:endr]
   integer ,         intent(in) :: lnumr               ! local number of cells
...
! check bounds of arrays
   SHR_ASSERT_ALL((ubound(fthresh) == (/endr/)),         errMsg(__FILE__, __LINE__))
   SHR_ASSERT_ALL((ubound(gindex)  == (/endr/)),         errMsg(__FILE__, __LINE__))
   SHR_ASSERT_ALL((ubound(evel)    == (/endr, nt_rtm/)), errMsg(__FILE__, __LINE__))
...
```

b. **Pointer** - We are moving away from using pointers as a way pass information into routines. We still allow this but with the use of an associate statement.

Comments and Pointer Intent

```
associate(&
   ptrIn    => derivedType%readOnlyInRoutine     , & ! [character(len=8) (:)] intent = in used for
x
   ptrInOut => derivedType%readAndWriteInRoutine , & ! [real(r8) (:,:,:)]     intent = inout used
for y
   ptrOut   => derivedType%writeOnlyInRoutine      & ! [real(r8) (:)]         intent = out used
for z
)
...
end associate
```

c. **Global data** - We are moving away from using global data as a way pass information into routines. Unless you are extending existing functionality (e.g. using a namelist variable to control some execution) we will generally not allow any new use of a global variable. The other exception to this rule is that it is OK to you use global type declarations.

Global Data. Global data in CanopyFluxesMod.F90

```
use clm_varctl  , only: iulog, use_cn, use_lch4, use_c13, use_c14, use_cndv
!
   if ( use_c13 ) then
      c13o2(p) = forc_pc13o2(g)
   endif ! use_c13
!
```

Global Data. Using a global type declaration

```
use decompMod   , only : bounds_type ! using a global type declaration
...
subroutine CNSoilLittVertTransp(bounds, num_soilc, filter_soilc)
...
type(bounds_type), intent(in) :: bounds  ! bounds
...
```

Indentation

1. Indent consistently and make sure your auto-tab function in your editor converts tabs to spaces.
    a. Indent of most blocks (do, if, while, etc.): 3 spaces
    b. Indent of "program" blocks (program, module, subroutine, function): 2 spaces
    c. Indent of "associate" blocks: 0 spaces
    d. Indent of continuation lines: 5 (this can vary based on look and feel)

Indentation

```
! a dash, '-', represents a space
--use thisModule , only : thisRoutine
!
--subroutine foo
----use thatModule , only : thatRoutine
----associate ( &
-------ptr1 => someDerivedType%ptr1 , &
-------ptr2 => someDerivedType%ptr2   &
----)
----if ( someLogical ) then
-------x=x+1
! leave a line before and after statements that stand alone.
-------if (x == 0) then endrun()
! leave a line before and after statements that stand alone.
-------do while ( x <= 1000 )
----------! do some stuff
----------z = 1.0_r8 / radius
-------end do ! end x<=1000
----endif ! someLogical
!
----end associate
--end subroutine
```

2. Do not leave hanging white space at the end of any line of code; git diff shows this in red spaces.
3. Do not tabularize code by adding extra spaces to any line of code. It may be tempting to do this for better readability, but it also takes time to manage code that needs realignment when new code comes in.

Statements and Subroutines

1. Don't repeat yourself. If you are copying and pasting code, then write a subroutine or function to encapsulate functionality.

Example of repeated code - Do

```
!
! an example of using a subprogram to remove repeated code where
! the max daylength is calculated in a function.
!
! From iniTimeConst:

      ! initialize maximum daylength, based on latitude and maximum declination
      ! maximum declination hardwired for present-day orbital parameters,
      ! ± 23.4667 degrees = ± 0.409571 radians, use negative value for S. Hem
      max_decl = 0.409571
      if (grc%lat(g) .lt. 0._r8) max_decl = -max_decl
         max_dayl(c) = calcMaxDayFunc( grc%lat(g) , max_decl )

! From CanopyFluxesMod:

      ! calculate daylength
      dayl(c) = calcMaxDayFunc( lat(g) , decl(c) )

! From CNPhenologyMod (copied & pasted in two places in this module):

         lat = (SHR_CONST_PI/180._r8)*grc%latdeg(pgridcell(p))
         dayl(p) = calcMaxDayFunc( lat , decl(c) )
```

Example of repeated code - Don't do

```
!
! an example of what not to do in terms of repeating code.
! The max daylength calculation is repeated.
!
! From iniTimeConst:

      ! initialize maximum daylength, based on latitude and maximum declination
      ! maximum declination hardwired for present-day orbital parameters,
      ! ± 23.4667 degrees = ± 0.409571 radians, use negative value for S. Hem
      max_decl = 0.409571
      if (grc%lat(g) .lt. 0._r8) max_decl = -max_decl
         temp = -(sin(grc%lat(g))*sin(max_decl))/(cos(grc%lat(g)) * cos(max_decl))
         temp = min(1._r8,max(-1._r8,temp))
         max_dayl(c) = 2.0_r8 * 13750.9871_r8 * acos(temp)

! From CanopyFluxesMod:

      ! calculate daylength
      temp = -(sin(lat(g))*sin(decl(c)))/(cos(lat(g)) * cos(decl(c)))
      temp = min(1._r8,max(-1._r8,temp))
      dayl = 2.0_r8 * 13750.9871_r8 * acos(temp)

! From CNPhenologyMod (copied & pasted in two places in this module):

         lat = (SHR_CONST_PI/180._r8)*grc%latdeg(pgridcell(p))
         temp = -(sin(lat)*sin(decl(c)))/(cos(lat) * cos(decl(c)))
         temp = min(1._r8,max(-1._r8,temp))
         dayl(p) = 2.0_r8 * 13750.9871_r8 * acos(temp)
```

2. Fortran allows you to place multiple statements on one line. Don't do this.

Multiple Statements - Do

```
! do this:
x = 0
y = 0
z = 0
```

Multiple Statements - Don't do

```
! not this:
x = 0 ; y=0; z=0
```

3. Use temporary variables often

Temporary Variables - Do

```
! do this:
someFactor = ( const1/const2 ) * const3
additiveFluxes = ( add1Flx + add2Flx + add3Flx ) ** someFactor
subFluxes = (sub1Flx + sub2Flx + sub3Flx) ** someFactor
sumVal = additiveFluxes + subFluxes
```

Temporary Variables - Don't do

```
! not this:
sumVal = (( add1Flx + add2Flx + add3Flx ) ** (( const1/const2 ) * const3 )) + ((sub1Flx + sub2Flx +
sub3Flx) ** (( const1/const2 ) * const3 ))
```

4. Keep subroutines short and organize the code into the smallest reasonable chunk of code that can stand on its own. Basically, you're looking for dividing routines into logical units of work. For example, don't put initialization, some science and then diagnostics all in one routine, these would be best divided into 4 routines; in this case one routine that calls three additional routines.
    - Use parenthesis to clarify you statements even when the Fortran precedence rules will evaluate items in the correct order.

Preprocessor Macros (CPP Tokens)

- We are in the process of removing CPP Tokens from all of CLM (see our refactoring page) and will not accept code using them.

Preprocessor Macros (CPP Tokens) - Do

```
! do this:
if (useNewPhysics == .true.) then
   ...
   ! do some new physics
   ...
endif
```

Preprocessor Macros (CPP Tokens) - Don't do

```
! not this:
#if (defined USENEWPHYSICS)
   ...
   ! do some new physics
   ...
#endif
```