# NEMO coding conventions

– version 3 –

NEMO System Team

March 2011

# Contents

# 1 Introduction

This document describes conventions used in NEMO coding and suggested for its development. The objectives are to offer a guide to all readers of the NEMO code, and to facilitate the work of all the developers, including the validation of their developments, and eventually the implementation of these developments within the NEMO platform.
A first approach of these rules can be found in the code in $NEMO/OPA\_SRC/module\_example$ where all the basics coding conventions are illustrated. More details can be found below.
This work is based on the coding conventions in use for the Community Climate System Model, [1] the previous version of this document ("FORTRAN coding standard in the OPA System") and the expertise of the NEMO System Team which can be contacted for further information ($nemo\_st@locean - ipsl.upmc.fr$) After a general overview below, this document will describe :

- The style rules, i.e. the syntax, appearance and naming conventions chosen to improve readability of the code;
- The content rules, i.e. the conventions to improve the reliability of the different parts of the code;
- The package rules to go a step further by improving the reliability of the whole and interfaces between routines and modules.

# 2 Overview and general conventions

NEMO has several different components: ocean dynamics ($OPA\_SRC$), sea-ice ($LIM\_SRC$), ocean biogeochemistry- ($TOP\_SRC$), linear-tangent and adjoint of the dynamics ($TAM$) each of them corresponding to a directory. In each directory, one will find some FORTRAN files and/or subdirectories, one per functionality of the code: $BDY$ (boundaries), $DIA$ (diagnostics), $DOM$ (domain), $DYN$ (dynamics), $LDF$ (lateral diffusion), etc...
All name are chosen to be as self-explanatory as possible, in English, all prefixes are 3 digits.
English is used for all variables names, comments, and documentation.
Physical units are MKS. The only exception to this is the temperature, which is expressed in degrees Celsius, except in bulk formulae and part of LIM sea-ice model where it is in Kelvin. See $DOM/phycst.F90$ files for conversions.

# 3 Architecture

Within each directory, organisation of files is driven by orthogonality, i.e. one functionality of the code is intended to be in one and only one directory, and one module and all its related routines are in one file. The functional modules are:

- SBC surface module
- IOM management of the I/O
- NST interface to AGRIF (nesting model) for dynamics and biogeochemistry
- OBC, BDY management of structured and unstructured open boundaries
- C1D 1D (vertical) configuration for dynamics, sea-ice and biogeochemistry
- OFF off-line module: passive tracer or biogeochemistry alone
- CFG tutorial and reference configurations
- DOC documentation

For example, the file $domain.F90$ contains the module $domain$ and all the subroutines related to this module ($dom\_init, dom\_nam, dom\_ctl$).

[1] http://www.cesm.ucar.edu/working_groups/Software/dev_guide/dev_guide/node7.html

# 4 Style rules

## 4.1 Argument list format

Routine argument lists will contain a maximum 5 variables per line, whilst continuation lines can be used. This applies both to the calling routine and the dummy argument list in the routine being called. The purpose is to simplify matching up the arguments between caller and callee.

```
SUBROUTINE tra_adv_eiv( kt, pun, pvn, pwn )

      CALL tra_adv_eiv( kt, zun, zvn, zwn )
```

## 4.2 Array syntax

Except for long loops (see below), array notation should be used if possible. To improve readability the array shape must be shown in brackets, e.g.:

```
onedarraya(:) = onedarrayb(:) + onedarrayc(:)
twodarray (:,:) = scalar * anothertwodarray(:,:)
```

When accessing sections of arrays, for example in finite difference equations, do so by using the triplet notation on the full array, e.g.:

```
twodarray(:,2:len2) = scalar                        &
   &                * ( twodarray2(:,1:len2-1 )     &
   &                -   twodarray2(:,2:len2 ) )
```

For long, complicated loops, explicitly indexed loops should be preferred. In general when using this syntax, the order of the loops indices should reflect the following scheme (for best usage of data locality):

```
DO jk = 1, jpk
   DO jj = 1, jpj
      DO ji = 1, jpi
         array(ji,jj,jk) = ...
      END DO
   END DO
END DO
```

## 4.3 Case

All FORTRAN keywords are in capital :

```
 DIMENSION, WRITE, DO, END DO, NAMELIST
```

All other parts of the NEMO code will be written in lower case.

## 4.4 Comments

Comments in the code are useful when reading the code and changing or developing it.
The full documentation and detailed explanations are to be added in the reference manual (TeX files, aside from the code itself).
In the code, the comments should explain variable content and describe each computational step.
Comments in the header start with "!!". For more details on the content of the headers, see Content rules/Headers in this document.
Comments in the code start with "!".
All comments are indented (3, 6, or 9 blank spaces).
Short comments may be included on the same line as executable code, and an additional line can be used with proper alignment. For example:

```
      zx = zx *zzy   ! Describe what is going on and if it is
      !              ! too long use another ! for proper
      !              ! alignment with automatic indentation
```

More in-depth comments should be written in the form:

```
!   Check of some namelist values
```

or

```
!
!         !<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
!         !  Bottom boundary condition on tke
!         !<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
!
```

Key features of this style are 1) it starts with a "!" in the column required for proper indentation, 2) the text is offset above and below by a blank line or a content line built for underlying.

## 4.5 Continuation lines

Continuation lines can be used with precise alignment for readability. For example:

```
avmu(ji,jj,jk) = avmu(ji,jj,jk) * ( un(ji,jj,jk-1) - un(ji,jj,jk) )    &
   &                             * ( ub(ji,jj,jk-1) - ub(ji,jj,jk) )    &
   &            / (  fse3uw_n(ji,jj,jk)          &
   &                 * fse3uw_b(ji,jj,jk) )
```

Code lines, which are continuation lines of assignment statements, must begin to the right of the column of the assignment operator. Due to the possibility of automatic indentation in some editor (emacs for example), use a "&" as first character of the continuing lines to maintain the alignment.

## 4.6 Declaration of arguments and local variables

In a routine, input arguments and local variables are declared 1 per line, with a comment field on the same line as the declaration. Multiple comment lines describing a single variable are acceptable if needed. For example:

```
INTEGER             ::   kstp   ! ocean time-step index
```

## 4.7 F90 Standard

NEMO software adheres to the FORTRAN 95 language standard and does not rely on any specific language or vendor extensions.

## 4.8 Free-Form Source

Free-form source will be used. The F90/95 standard allows lines of up to 132 characters, but a self-imposed limit of 80 should enhance readability, or print source files with two columns per page. Multi-line comments that extend to column 100 are unacceptable.

## 4.9 Indentation

Code as well as comment lines within loops, if-blocks, continuation lines, MODULE or SUBROUTINE statements will be indented 3 characters for readability. (except for CONTAINS that remains at first column)

```
MODULE mod1
   REAL(wp) xx
CONTAINS
   SUBROUTINE sub76( px, py, pz, pw, pa,   &
      &              pb, pc, pd, pe      )
      <instruction>
   END SUBROUTINE sub76
END MODULE mod1
```

## 4.10 Loops

Loops, if explicit, should be structured with the do-end do construct as opposed to numbered loops. Nevertheless non-numeric labels can be used for a big iterative loop of a recursive algorithm. In the case of a long loop, a self-descriptive label can be used (i.e. not just a number).

## 4.11 Naming Conventions: files

A file containing a module will have the same name as the module it contains (because dependency rules used by "make" programs are based on file names). [2]

## 4.12 Naming Conventions: modules

Use a meaningful English name and the "3 letters" naming convention: first 3 letters for the code section, and last 3 to describe the module. For example, zdftke, where "zdf" stands for vertical diffusion, and "tke" for turbulent kinetic energy.
Note that by implication multiple modules are not allowed in a single file. The use of common blocks is deprecated in Fortran 90 and their use in NEMO is strongly discouraged. Modules are a better way to declare static data. Among the advantages of modules is the ability to freely mix data of various types, and to limit access to contained variables through the use of the ONLY and PRIVATE attributes.

## 4.13 Naming Conventions: variables

All variable should be named as explicitly as possible in English. The naming convention concerns prefix letters of these name, in order to identify the variable type and status.
Never use a FORTRAN keyword as a routine or variable name.
The table below lists the starting letter(s) to be used for variable naming, depending on their type and status:

| Type / Status | integer | real | logical | character | double precision | complex |
|---|---|---|---|---|---|---|
| public or module variable | **m n** *but not* **nn_** | **a b e f g h o q** *to* **x** but not **fs rn_** | **l** *but not* **lp ld ll ln_** | **c** *but not* **cp cd cl cn_** | **d** *but not* **dp dd dl dn_** | **y** *but not* **yp yd yl** |
| dummy argument | **k** *but not* **kf** | **p** *but not* **pp pf** | **ld** | **cd** | **dd** | **yd** |
| local variable | **i** | **z** | **ll** | **cl** | **cd** | **yl** |
| loop control | **j** *but not* **jp** | | | | | |
| parameter | **jp** | **pp** | **lp** | **cp** | **dp** | **yp** |
| namelist | **nn_** | **rn_** | **ln_** | **cn_** | **dn_** | |
| CPP macro | **kf** | **sf** | | | | |

## 4.14 Operators

Use of the operators $<, >, <=, >=, ==, /=$ is strongly recommended instead of their deprecated counterparts, $lt., .gt., .le., .ge., .eq., and.ne.$ The motivation is readability. In general use the notation:
$< Blank >< Operator >< Blank >$

[2] For example, if routine A "USE"s module B, then "make" must be told of the dependency relation which requires B to be compiled before A. If one can assume that module B resides in file B.o, building a tool to generate this dependency rule (e.g. A.o: B.o) is quite simple. Put another way, it is difficult (to say nothing of CPU-intensive) to search an entire source tree to find the file in which module B resides for each routine or module which "USE"s B.

### 4.15 Pre processor

Where the use of a language pre-processor is required, it will be the C pre-processor (cpp).
The cpp key is the main feature used, allowing to ignore some useless parts of the code at compilation step.
The advantage is to reduce the memory use; the drawback is that compilation of this part of the code isn't checked.
The cpp key feature should only be used for a few limited options, if it reduces the memory usage. In all cases, a logical variable and a FORTRAN $IF$ should be preferred. When using a cpp key $key\_optionname$, a corresponding logical variable $lk\_optionname$ should be declared to allow FORTRAN $IF$ tests in the code and a FORTRAN module with the same name (i.e. $optionname.F90$) should be defined. This module is the only place where a "#if defined" command appears, selecting either the whole FORTRAN code or a dummy module. For example, the TKE vertical physics, the module name is $zdftke.F90$, the CPP key is $key\_zdftke$ and the associated logical is $lk\_zdftke$.

The following syntax:

```
#if defined key_optionname
!! Part of code conditionally compiled if cpp key key_optionname is active
#endif
```

Is to be used rather than the #ifdef abbreviate form since it may have conflicts with some Unix scripts.

Tests on cpp keys included in NEMO at compilation step:

- The CPP keys used are compared to the previous list of cpp keys (the compilation will stop if trying to specify a non-existing key)
- If a change occurs in the CPP keys used for a given experiment, the whole compilation phase is done again.

# 5 Content rules

## 5.1 Configurations

The configuration defines the domain and the grid on which NEMO is running. It may be useful to associate a cpp key and some variables to a given configuration, although the part of the code changed under each of those keys should be minimized. As an example, the "ORCA2" configuration (global ocean, 2 degrees grid size) is associated with the cpp key $key\_orca2$ for which

```
cp_cfg = "orca"
jp_cfg = 2
```

## 5.2 Constants

Physical constants (e.g. pi, gas constants) must never be hardwired into the executable portion of a code. Instead, a mnemonically named variable or parameter should be set to the appropriate value, in the setup routine for the package. We realize than many parameterizations rely on empirically derived constants or fudge factors, which are not easy to name. In these cases it is not forbidden to leave such factors coded as "magic numbers" buried in executable code, but comments should be included referring to the source of the empirical formula. Hard-coded numbers should never be passed through argument lists.

## 5.3 Declaration for variables and constants

### 5.3.1 Rules :

Variables used as constants should be declared with attribute PARAMETER and used always without copying to local variables, inorder to prevent from using different values for the same constant or changing it accidentally.

- Usage of the DIMENSION statement or attribute is required in declaration statements
- The "::" notation is quite useful to show that this program unit declaration part is written in standard FORTRAN syntax, even if there are no attributes to clarify the declaration section. Always use the notation <blank>::<three blanks> to improve readability.
- Declare the length of a character variable using the CHARACTER (len=xxx) syntax [3]

[3]The len specifier is important because it is possible to have several kinds for characters (e.g. Unicode using two bytes per character, or there might be a different kind for Japanese e.g. NEC).

- For all global data (in contrast to module data, that is all data that can be access by other module) must be accompanied with a comment field on the same line. [4]
  For example:

```
REAL(wp), DIMENSION(jpi,jpj,jpk) ::  ua  &  !: i-horizontal velocity (m/s)
```

#### 5.3.2 Implicit None:

All subroutines and functions will include an IMPLICIT NONE statement. Thus all variables must be explicitly typed. It also allows the compiler to detect typographical errors in variable names. For modules, one IMPLICIT NONE statement in the modules definition section is needed. This also removes the need to have IMPLICIT NONE statements in any routines that are CONTAIN'd in the module. Improper data initialisation is another common source of errors. [5] To avoid problems, initialise variables as close as possible to where they are first used.

#### 5.3.3 Attributes:

$PRIVATE/PUBLIC$ : All resources of a module are $PUBLIC$ by default. A reason to store multiple routines and their data in a single module is that the scope of the data defined in the module can be limited to the routines which are in the same module. This is accomplished with the $PRIVATE$ attribute.
$INTENT$ : All dummy arguments of a routine must include the $INTENT$ clause in their declaration in order to improve control of variables in routine calls.

## 5.4 Headers

Prologues are not used in NEMO for now, although it may become an interesting tool in combination with ProTeX auto documentation script in the future. Rules to code the headers and layout of a module or a routine are illustrated in the example module available with the code : *NEMO/OPA_SRC/module_example*

## 5.5 Interface blocks

Explicit interface blocks are required between routines if optional or keyword arguments are to be used. They also allow the compiler to check that the type, shape and number of arguments specified in the CALL are the same as those specified in the subprogram itself. FORTRAN 95 compilers can automatically provide explicit interface blocks for routines contained in a module.

## 5.6 I/O Error Conditions

I/O statements which need to check an error condition will use the $iostat =< integervariable >$ construct instead of the outmoded end= and err=.
Note that a 0 value means success, a positive value means an error has occurred, and a negative value means the end of record or end of file was encountered.

## 5.7 PRINT - ASCII output files

Output listing and errors are directed to $numout$ logical unit =6 and produces a file called $ocean.output$ (use ln_prt to have one output per process in MPP). Logical $lwp$ variable allows for less verbose outputs. To output an error from a routine, one can use the following template:

```
IF( nstop /= 0 .AND. lwp ) THEN   ! error print
   WRITE(numout,cform_err)
   WRITE(numout,*) nstop, ' error have been found'
ENDIF
```

[4] This allows a easy research of where and how a variable is declared using the unix command: "grep var *90 —grep !:".

[5] A variable could contain an initial value you did not expect. This can happen for several reasons, e.g. the variable has never been assigned a value, its value is outdated, memory has been allocated for a pointer but you have forgotten to initialise the variable pointed to.

### 5.8 Precision

Parameterizations should not rely on vendor-supplied flags to supply a default floating point precision or integer size. The F95$KIND$ feature should be used instead. In order to improve portability between 32 and 64 bit platforms, it is necessary to make use of kinds by using a specific module ($OPA\_SRC/par\_kind.F90$) declaring the "kind definitions" to obtain the required numerical precision and range as well as the size of INTEGER. It should be noted that numerical constants need to have a suffix of $\_kindvalue$ to have the according size.
Thus $wp$ being the "working precision" as declared in $OPA\_SRC/par\_kind.F90$, declaring real array $zpc$ will take the form:

```
      REAL(wp), DIMENSION(jpi,jpj,jpk) ::  zpc      ! power consumption
```

### 5.9 Structures

The TYPE structure allowing to declare some variables is more often used in NEMO, especially in the modules dealing with reading fields, or interfaces.For example

```
 ! Definition of a tracer as a structure
      TYPE PTRACER
         CHARACTER(len = 20)  :: sname  !: short name
         CHARACTER(len = 80 ) :: lname  !: long name
         CHARACTER(len = 20 ) :: unit   !: unit
         LOGICAL              :: lini   !: read in a file or not
         LOGICAL              :: lsav   !: ouput the tracer or not
      END TYPE PTRACER

      TYPE(PTRACER) , DIMENSION(jptra) :: tracer
```

Missing rule on structure name??

## 6 Packages coding rules

### 6.1 Bounds checking

NEMO is able to run when an array bounds checking option is enabled (provided the cpp key $key\_vectopt\_loop$ is not defined).
Thus, constructs of the following form are disallowed:

```
REAL(wp) :: arr(1)
```

where "arr" is an input argument into which the user wishes to index beyond 1. Use of the (*) construct in array dimensioning is forbidden also because it effectively disables array bounds checking.

### 6.2 Communication

A package should refer only to its own modules and subprograms and to those intrinsic functions included in the Fortran standard.
All communication with the package will be through the argument list or namelist input. [6]

### 6.3 Error conditions

When an error condition occurs inside a package, a message describing what went wrong will be printed (see PRINT - ASCII output files). The name of the routine in which the error occurred must be included. It is acceptable to terminate execution within a package, but the developer may instead wish to return an error flag through the argument list, see $stpctl.F90$.

[6] The point behind this rule is that packages should not have to know details of the surrounding model data structures, or the names of variables outside of the package. A notable exception to this rule is model resolution parameters. The reason for the exception is to allow compile-time array sizing inside the package. This is often important for efficiency.

## 6.4 Memory management

The main action is to identify and declare which arrays are PUBLIC and which are PRIVATE.
As of version 3.3.1 of NEMO, the use of static arrays (size fixed at compile time) has been deprecated. All module arrays are now declared ALLOCATABLE and allocated in either the <module_name>_alloc() or <module_name>_init() routines. The success or otherwise of each ALLOCATE must be checked using the $Stat =< integer\ variable >$ optional argument.

In addition to arrays contained within modules, many routines in NEMO require local, "workspace" arrays to hold the intermediate results of calculations. In previous versions of NEMO, these arrays were declared in such a way as to be automatically allocated on the stack when the routine was called. An example of an automatic array is:

```
SUBROUTINE sub(n)
   REAL :: a(n)
   ...
END SUBROUTINE sub
```

The downside of this approach is that the program will crash if it runs out of stack space and the reason for the crash might not be obvious to the user.

Therefore, as of version 3.3.1, the use of automatic arrays is deprecated. Instead, a new module, "wrk_nemo," has been introduced which contains 1-,2-,3- and 4-dimensional workspace arrays for use in subroutines. These workspace arrays should be used in preference to declaring new, local (allocatable) arrays whenever possible. The only exceptions to this are when workspace arrays with lower bounds other than 1 and/or with extent(s) greater than those in the *wrk_nemo* module are required.

The 2D, 3D and 4D workspace arrays in *wrk_nemo* have extents $jpi$, $jpj$, $jpk$ and $jpts$ ($x$, $y$, $z$ and tracers) in the first, second, third and fourth dimensions, respectively. The 1D arrays are allocated with extent MAX($jpi \times jpj, jpk \times jpj, jpi \times jpk$).

The REAL (KIND=$wp$) workspace arrays in *wrk_nemo* are named e.g. $wrk\_1d\_1$, $wrk\_4d\_2$ etc. and should be accessed by USE'ing the *wrk_nemo* module. Since these arrays are available to any routine, some care must be taken that a given workspace array is not already being used somewhere up the call stack. To help with this, *wrk_nemo* also contains some utility routines; *wrk_in_use()* and *wrk_not_released()*. The former first checks that the requested arrays are not already in use and then sets internal flags to show that they are now in use. The *wrk_not_released()* routine un-sets those internal flags. A subroutine using this functionality for two, 3D workspace arrays named $zwrk1$ and $zwrk2$ will look something like:

```
SUBROUTINE sub()
   USE wrk_nemo, ONLY: wrk_in_use, wrk_not_released
   USE wrk_nemo, ONLY: zwrk1 => wrk_3d_5, zwrk2 => wrk_3d_6
   !
   IF(wrk_in_use(3, 5,6)THEN
      CALL ctl_stop('sub: requested workspace arrays unavailable.')
      RETURN
   END IF
   ...
   ...
   IF(wrk_not_released(3, 5,6)THEN
      CALL ctl_stop('sub: failed to release workspace arrays.')
   END IF
   !
END SUBROUTINE sub
```

The first argument to each of the utility routines is the dimensionality of the required workspace (1–4). Following this there must be one or more integers identifying which workspaces are to be used/released. Note that, in the interests of keeping the code as simple as possible, there is no use of POINTERs etc. in the *wrk_nemo* module. Therefore it is the responsibility of the developer to ensure that the arguments to *wrk_in_use()* and *wrk_not_released()* match the workspace arrays actually being used by the subroutine.

If a workspace array is required that has extent(s) less than those of the arrays in the *wrk_nemo* module then the advantages of implicit loops and bounds checking may be retained by defining a pointer to a sub-array as follows:

```
SUBROUTINE sub()
   USE wrk_nemo, ONLY: wrk_in_use, wrk_not_released
   USE wrk_nemo, ONLY: wrk_3d_5
   !
   REAL(wp), DIMENSION(:,:,:), POINTER :: zwrk1
   !
   IF(wrk_in_use(3, 5)THEN
      CALL ctl_stop('sub: requested workspace arrays unavailable.')
      RETURN
   END IF
   !
   zwrk1 => wrk_3d_5(1:10,1:10,1:10)
   ...
END SUBROUTINE sub
```

Here, instead of "use associating" the variable $zwrk1$ with the array $wrk\_3d\_5$ (as in the first example), it is explicitly declared as a pointer to a 3D array. It is then associated with a sub-array of $wrk\_3d\_5$ once the call to *wrk_in_use()* has completed successfully. Note that in F95 (to which NEMO conforms) it is not possible for either the upper or lower array bounds of the pointer object to differ from those of the target array.

In addition to the REAL (KIND=$wp$) workspace arrays, *wrk_nemo* also contains 2D integer arrays and 2D REAL arrays with extent ($jpi$, $jpk$), *i.e.* $xz$. The utility routines for the integer workspaces are *iwrk_in_use()* and *iwrk_not_released()* while those for the $xz$ workspaces are *wrk_in_use_xz()* and *wrk_not_released_xz()*.

Should a call to one of the *wrk_in_use()* family of utilities fail, an error message is printed along with a table showing which of the workspace arrays are currently in use. This should enable the developer to choose alternatives for use in the subroutine being worked on.

When compiling NEMO for production runs, the calls to *wrk_in_use()*/*wrk_not_released()* can be reduced to stubs that just return .FALSE. by setting the cpp key *key_no_workspace_check*. These stubs may then be inlined (and thus effectively removed altogether) by setting appropriate compiler flags (e.g. "-finline" for the Intel compiler or "-Q" for the IBM compiler).

## 6.5 Optimisation

Considering the new computer architecture, optimisation cannot be considered independently from the computer type. In NEMO, portability is a priority, before any too specific optimisation. Some tools are available to help:
For vector computers:

- using $key\_vectopt\_loop$ allows to unroll a loop

## 6.6 Package attribute: $PRIVATE, PUBLIC, USE, ONLY$

Module variables and routines should be encapsulated by using the PRIVATE attribute. What shall be used outside the module can be declared PUBLIC instead. Use USE with the ONLY attribute to specify which of the variables, type definitions etc. defined in a module are to be made available to the using routine.

## 6.7 Parallelism: using MPI

NEMO is written in order to be able to run on one processor, or on one or more using MPI (i.e. activating the cpp key $key\_mpp\_mpi$. The domain decomposition divides the global domain in cubes (see NEMO reference manual). Whilst coding a new development, the MPI compatibility has to be taken in account (see $LBC/lib\_mpp.F90$) and should be tested. By default, the $x$-$z$ part of the decomposition is chosen to be as square as possible. However, this may be overriden by specifying the number of subdomains in latitude and longitude in the nammpp section of the namelist file.

# 7 Features to be avoided

The code must follow the current standards of FORTRAN and ANSI C. In particular, the code should not produce any WARNING at compiling phase, so that users can be easily alerted of potential bugs when some appear in their new developments. ). Below is a list of features to avoid:

- COMMON blocks (use the declaration part of MODULEs instead)
- EQUIVALENCE (use POINTERs or derived data types instead to form data structures)
- Assigned and computed GOTOs (use the CASE construct instead)
- Arithmetic IF statements ( use the block IF, ELSE, ELSEIF, ENDIF or SELECT CASE construct instead)
- Labeled DO constructs (use unlabeled END DO instead)
- FORMAT statements (use character parameters or explicit format- specifiers inside the READ or WRITE statement instead)
- GOTO and CONTINUE statement (use IF, CASE, DO WHILE, EXIT or CYCLE statements or a contained
- PAUSE
- ENTRY statements: a subprogram must only have one entry point.
- RETURN it is obsolete and so not necessary at the end of program units
- STATEMENT FUNCTION
- Avoid functions with side effects. [7]
- DATA and BLOCK DATA - (use initialisers)

[7] First, the code is easier to understand, if you can rely on the rule that functions don't change their arguments, second, some compilers generate more efficient code for PURE (in FORTRAN 95 there are the attributes PURE and ELEMENTAL) functions, because they can store the arguments in different places. This is especially important on massive parallel and as well on vector machines.