US Department of Commerce
National Oceanic and Atmospheric Administration (NOAA)
National Weather Service
National Centers for Environmental Prediction (NCEP)

Office Note 492
http://doi.org/10.7289/V5/ON-NCEP-492

National Centers for Environmental Prediction
Coding Standards
Version 2016a

Samuel Trahan[I], Eugene Mirvis[I], Jacob R. Carley[I], Paul van Delst[I], J.Abeles[I], H.Alves[I], S.Earle[N], M.Ek[E], M.Iredell[E], K.Menlove[R], A.Ostapenko[C], R.Padilla[I], I.Rivin[E], A.v/d.Westhuysen[I], R.Wobus[I], J.Woollen[I]

C=NOAA Corps Federal E=NOAA NCEP EMC Federal I=I.M.Systems Group
N=NOAA NCEP Central Operations Federal R=Redline Performance

September 19, 2016

https://vlab.ncep.noaa.gov/web/ncep-coding-standards

Executive Summary

This document lists the coding standards to be followed by operational codes and scripts at NCEP, and describes the organization and procedures necessary to implement, enforce, and update them.

Purpose and Scope

This document establishes a set of coding standards that will be applied to all code and scripts that are or will be in the NCEP production suite, and code or scripts used for development of improvements to that suite. These standards are not meant to be immutable; we propose a Coding Standards Group to meet as needed to update them. This group will analyze feedback from developers throughout NCEP and among its collaborators, and use that feedback to drive standards updates. However, these standards will apply to all code and scripts, including ones presently in existence that do not follow the standards yet. For implementation packages that do not follow the standards yet, we propose a system of conditional, limited-term, limited-scope, exemptions based on a cost-benefit analysis until each package conforms with all standards.

How to Read this Document

The purpose of this document is to provide a list of standards. The standards are in sections whose name begins with a designator like CC-12 or CC-12-3 where “CC” is the category, and 12 or 12-3 identifies the requirement. Sections whose titles do not begin with a designator, such as this section and the previous, are NOT standards. Each section with a designator such as CC-12-3 contains the rule as well as any relevant justification, documentation references, and examples.

Example rule:

XX-12-3 Example of a Rule Heading

Rule text appears in the standard font for this document. This text defines the rule to be followed.

Bold text in boxes is for emphasis. **This text may restate the above requirement.**

Justification [and/or Clarification]:
This text explains why the rule exists or explains the context of the rule. Justification does not consist of rules to be followed; it merely provides clarification.

Example:
Gives examples to further clarify the rule. This section provides:

- examples of use that follows the rules, and/or
- examples of use that breaks the rules.

There may be code examples too. Frequently there are two code blocks: one that breaks the rule and one that follows it. The problematic part of the code is highlighted in red.

```
Print *,"Red or black monospaced, indented text is source code."
```

Coding Standards Group

These standards call for the creation of an ongoing NCEP Coding Standards Group, whose duty is to analyze coding standards and update them as needed to handle new languages, new language versions, new technologies, and other unexpected developments. Furthermore, this group will analyze requests to exempt a project from one or more of the standards discussed herein, and provide feedback to managers about whether that exemption should be made. Later sections discuss this in more detail.

Existing Codebase

NCEP runs codes which are written in a variety of languages, many of those codes were written according to old standards, or one at all. Updating or refactoring all legacy codes to adhere to the newly derived Environmental Equivalence (EE) coding standards would be burdensome and likely come at the cost of project development (e.g. forecast skill improvements, or financial cost). Furthermore, many so-called legacy codes are still under active development and maintenance.

However, there have been instances in which poor code harmed the forecast skill, delayed implementations, or risked the security of operational computers. Such problems can be avoided by improving all NCEP code quality to meet industry standards. We argue that it is long past time to enforce good coding practices at NCEP. However, we need a reasonable timeline for doing so, and that may have to be set differently on a per-project basis, depending on the codebase size and the number of developers available to that project.

For example, if a developer modifies a code, changing a line or subroutine, to what extent is the developer required to adhere to the EE coding standards? Does the developer have to modify the entire codebase to comply, or just the parts that changed? What if there is only one developer, tasked half-time, to maintain a legacy 500,000 line codebase?

To what extent should the coding standards be applied to community codes not maintained by NCEP? Many such projects may have external repositories and possibly their own standards. As is the case with legacy code, it may be burdensome to enforce the Environmental Equivalence coding standards upon incoming community code and would likely impede research to operations (R2O) activities. Is it sufficient to accept those community projects' own coding standards?

Our approach to solving these problems is based on a principle of limited exemptions. Project managers may request a limited-time exemption from specific rules in this document. They must back that request with a peer-reviewed, cost-benefit analysis. The ultimate decision of whether to approve the exemption will be made by the NCEP Director or delegate, based on a recommendation by the Coding Standards Group after evaluation of the cost-benefit analysis.

Table of Contents

SG: Standards Governance

SG-01 Scope of Standards

SG-01-1 All Standards Apply Universally

Every line of code in the production suite shall conform to every standard described in this document, unless granted an exemption. There are no “legacy codes” or “third-party codes” that do not need to conform.

SG-01-2 Standards Are Enforced Before the Change Control Board Meeting

Standards shall be enforced early in the implementation process, before the Change Control Board meeting.

SG-02 Principle of Limited Exemptions

The solution to previously mentioned issues, of transitioning to the new standards, is a system of limited-time, limited-scope exemptions that are peer-reviewed and approved by management based on a cost-benefit analysis described herein.

SG-02-1 Initial One Year Exemption

All implemented packages have an exemption to all rules in this document for one year starting at the date of this proposal’s acceptance as a standard; code handed off to NCO after one year is expected to meet the standards.

This rule does not grant any exemption to any other rules external to this document. That includes, but is not limited to, the NCEP Production Standards, the Environmental Equivalence standards, NOAA security regulations or any other regulations, laws, directives, treaties, or the conservation of energy.

SG-02-2 Requesting Additional Exemptions

To extend the one year exemption, the project shall provide an objective cost-benefit analysis that shows the cost of following particular standards exceed the benefits. The request specifies a time period (provisionally until the next production implementation), the specific list of rules (CC-12-3 syntax) to which the project is to be exempted, and the exact regions of the code in the project's codebase that need the exemption.

The Coding Standards Group shall provide an evaluation of the cost-benefit analysis and codebase, and a recommendation to management for or against an exemption within five business days of receiving the exemption request.

SG-02-3 Steps to Standards Process

Developers of production codes shall follow this four-step standards review process before the code is handed off to NCO:

1. New code development is scheduled for an implementation by a development group.
2. An unbiased subgroup, designated by the Coding Standards Group, reviews the code to see if standards are met, perhaps assisted by automatic tools.
3. If the standards are met, the process is complete. If it is not practical to meet all standards, the development group provides a cost-benefit analysis to ask for exemption. If standards are not met and an exemption is not to be requested, the code shall be corrected and resubmitted for review.
4. If necessary, the coding standards committee reviews the analysis and decides whether to recommend exemption for this implementation.

SG-02-4 NCEP Director or Delegate Grants Exemptions

The NCEP director, or an official who he or she delegates, shall make the final decision about exemptions, based on the recommendation of the Coding Standards Group.

SG-02-5 Exemption, Rule and Review Scope

The exemption request is submitted for an entire package and is reviewed as a whole. It may contain exemptions for specific sub-parts of a package, such as libraries or even individual files. This list of exemptions for a package is reviewed as a whole, and must contain a cost-benefit analysis for all exemptions collectively, not just for individual sub-parts.

SG-03 Coding Standards Group

An NCEP Coding Standards Group shall be established to update the coding standards, or analyze new languages for proposed addition to the ruleset. This section describes the rules for that group.

SG-03-1 Group Membership

The group shall be made up of members tasked by their managers to perform this work.

SG-03-2 Review Codes for Compliance

When developers are ready to schedule the implementation of code, a subgroup of the Coding Standards Group is tasked with reviewing the code, checking for compliance to the coding standards. This will be done with automated tools to the greatest degree possible.

SG-03-3 Evaluate Exemptions

In cases where the cost of compliance would outweigh the benefits and management has granted exemptions, as explained in SG-02, such information shall be given to the Coding Standards Group to help improve the next revision of the standards. This review and cost-benefit analysis process provides the primary feedback to the group.

SG-03-4 Revise Standards

The group will meet at designated times during the year to review and revise standards. This group will managed by the the Coding Standards Working Group Chair, who will report to NCEP Leadership.

SG-03-5 New Languages

When requested by NCEP management or code managers, the Coding Standards Group shall evaluate new languages for addition to the rules described herein.

SG-03-6 Old Languages

When a language can no longer be properly supported in the production computing environment, NCEP management may request that it be deprecated; the use of that language in subsequent implementations would require an exemption, which will only be approved if the continuing use of the language is consistent with the support available and with security requirements.

SG-04 Published Rules

SG-04-1 Published in a Permanent Public Record

To ensure coding standards will be easy for developers to locate, all coding standards shall be published in a public location where it will be permanently available, and never modified, for the entire future lifetime of NCEP. The first version of this document shall be published as an NCEP Office Note.

SG-04-2 Versioned Document

Updates to the document shall be versioned publications identified by a year and a letter (2016a, 2016b, 2017a, etc.) Each updated version shall contain the entirety of the rules, not just the revised information.

The Coding Standards Group shall provide a guide to the significant changes in each version as it is released.

GC: General Coding Standards

This chapter discusses coding standards that apply to all code and scripts in operations, regardless of language, even in languages that do not have specific standards in this document. Later chapters clarify details or add rules for specific languages or families of languages.

GC-01 Stick to a Language Standard

The developer shall pick a specific version or set of versions of a language standard or set of language standards. This is identified as the *target language standard* in the rest of this section.

GC-01-1 Use International Standards if They Exist

If an *internationally recognized standards body* has a standard for the language, then the developer shall choose one such standard as the target language standard.

To restate: Code should never, under any circumstances, be designed to meet a non-standard, proprietary version of a language when a widely-used international standard exists.

Every code shall target a specific language standard from an internationally-recognized standards organization and conform to that standard. It is never acceptable to target a proprietary version of a language, such as the gfortran version of Fortran or Intel version of C. The only exception is when no international standard exists.

Examples:

Internationally Recognized Standards Bodies:

- International Organization for Standardization (ISO)
- International Electrotechnical Commission (IEC)

International Language Standards:

- ISO/IEC 9899 (the C language)
- ISO/IEC 14882 (the C++ language)
- ISO/IEC 1539-1: 2010 (Fortran 2008 base language).

Proprietary Standards (Not Allowed):

- GNU gfortran version of Fortran
- Intel version of C.

GC-01-2 Standards for Non-Standardized Languages

Some languages do not yet have standards made via internationally-recognized standards bodies. For such languages, the developer should pick a specific version that is well-documented by an organization that maintains it, and should verify with NCO that the chosen version is available or can be installed and supported.

Example:
No internationally-recognized standards bodies have standards for Python. Instead, one can choose Python 2.7 from python.org.

GC-01-3 Use Final Release Versions of Languages

Only final, public release versions of languages, shall be used.

International standards bodies release draft or final draft versions. Languages without international standards may have developmental or pre-release versions. Such versions are not allowed.

GC-01-4 No Deprecated, Broken, or Removed Features

Any language features that are deprecated or removed in the target language standard shall not be used in the code.

Examples:

- in the latest Fortran standards, computed **`goto`**, and in Fortran 90, fixed-form source files.
- In Python 2.6.6, the Python **`subprocess`** module which cannot launch multi-stage pipelines due to known bugs.

GC-01-5 No Implementation-Specific Features

Compiler-specific or operating-system-specific features shall not be used, except when absolutely necessary for portability or efficiency reasons. If used, the developer shall provide a second, standard-conforming, functionally equivalent, implementation.

In some cases, it may be critical to use an implementation-specific feature, such as for porting to a problematic compiler or to ensure fast execution. In those cases, the developer shall also provide a functionally equivalent, standard-conforming, implementation for portability purposes.

GC-01-6 Avoid Using Recent Features

Developers shall not use recent language features that are not widely supported. Later chapters detail which features are not allowed.

Justification:
Language features that have been standardized recently are typically not widely supported. Even if they are supported, the NOAA Security may not have been able to secure a recent enough version of the compiler or interpreter needed to use those features. The amount of time that it takes to support a language feature varies from language to language, so guidance on details of this is left to the language-specific chapters.

GC-01-7 Careful Use of Advanced Language Features

Some scripting languages have more advanced features than POSIX sh. For example, bash and ksh have types, arrays and regular expressions. Python and Perl add to that classes, exception handling, and lambda functions. Developers are encouraged to use these features so long as the benefits of that use outweigh the disadvantages. The language-specific chapters clarify the rules about these features.

GC-02 Program Exit

GC-02-1 Eight-Bit Integers

All exit codes shall fit within an eight-bit integer; within either 0 to 255 or -128 to 127, inclusive.

Justification:
POSIX uses eight bit integers to store exit codes from programs. Hence, all exit codes must be from 0 to 255, inclusive. It is also acceptable to use the equivalent two's complement signed integer range of -128 to 127, inclusive. However, the program must not use exit codes that require more than eight bits to represent such as 777, -999 or "failure."

GC-02-2 Exit Codes for Error Reporting

Programs that perform an operation and report its success shall exit with status 0 on success. In case of error, they should exit with well-defined, meaningful, eight-bit (see GC-02-1) error codes.

GC-02-3 Exit Codes for Numeric Information

As a special exception to GC-02-2, if the program provides numerical information via its exit code, it is acceptable to do so, as long as it fits within an eight-bit integer (see GC-02-1).

GC-02-4 Check Exit Codes

When executing a subprocess, the parent program or script shall check the exit code and react appropriately to failure statuses (see GC-02-2).

GC-02-5 Correctly Clean Multi-processing Environments Before Exit

Programs using multiprocessing or multithreading must ensure all processes and threads exit upon termination or abort.

Examples:

1. MPI programs must ensure all MPI ranks exit at the same time.
2. CPython threaded applications must wait for all threads to exit before the main program exits. Note that this requires special handling when programs receive signals.

GC-03 Source

GC-03-1 Code in English Using Printable Characters

All code shall be written using printable characters, tabs, end-of-lines and spaces. Identifiers (variable names, class names, etc.) shall be in English unless they are technical or scientific terms that have no English equivalent or interact with libraries that use non-English identifiers. However, in such cases, the terms shall be defined in English in the documentation.

Example:
The ratio of the circumference of a circle divided by its diameter, π, shall be given an English name such as pi with suitable documentation.

GC-03-2 English Comments and Documentation

Documentation and all code comments shall be available in English. It is acceptable to have translations available in other languages, so long as the English language code comments and documentation contain all of the information that is in other languages.

GC-03-3 Source Code Is Mandatory Unless Other Rules State Otherwise

Unless specified elsewhere in this document, compiled software used by NCEP shall be compiled by NCEP or its collaborators from human-readable source code on machines owned, rented by, or used in agreement with the US Government. Hence, it is unacceptable to use pre-compiled executables or libraries, unless allowed by GC-03-4. This includes languages that are bytecode-based such as Python and Java; source code is still mandatory.

GC-03-4 Source Code Not Mandatory for System Administrators or Support Contract

The only situations where source code is not mandatory are:

1. Programs and libraries provided under contract with the US Government where the contract supports such software installation and use in NCEP.
2. Programs and libraries installed by system administrators.

Note that this still forbids use of user-installed closed-source software, or pre-compiled software installed by a user from some other source.

Source code is mandatory. Compiled software used by NCEP shall be compiled by NCEP or its collaborators from human-readable source code on machines owned, rented by, or under contract with, the US Government. Only system administrators and US Government software support contracts are exempt from this rule.

GC-04 Documentation

GC-04-1 Main Program Documentation

The main program documentation, as viewed from outside shall provide the following information. This could be convey via a usage message, a unix “man” page, a separate manual document, ecFlow manual page, a website, or some other method. Such information does not have to be inside the code itself. Specifically, the following shall be provided:

1. Author list and contact information.
2. Meaning of program arguments.
3. Environment variables read by the program.
4. Purpose of the program.
5. Meaning of program exit codes.
6. Program input and output files.
7. Side-effects such as network access, file system metadata modification, user configuration changes, subprocess execution, or any other possible side-effect.

GC-04-2 Language-Specific Documentation Capabilities

Developers are encouraged to use language-specific or language-aware documentation capabilities.

Justification:

Such features connect the documentation to automatic help programs and allow automatic generation of manuals. They also allow documentation of features that this section cannot anticipate.

Examples:

- Perl POD
- Python docstrings
- Doxygen
- Javadoc

GC-05 Initialize Before Use

GC-05-1 No Uninitialized Storage

All memory, files, registers, or other storage areas that are not inherited from the parent process or passed from other processes shall be initialized before first read. Note that environment variables and command-line arguments are inherited from the parent process, and hence are exempted from this rule. Memory shared between processes shall be initialized by at least one process before any process reads it.

GC-05-2 Automatic Initialization is Allowed

In some cases, languages provide suitable default values or other automatic initialization mechanisms. Examples are Perl undef and Python None. In C++, local variables that are instances of a class have their default constructor called on them automatically upon declaration. Codes shall use these mechanisms only when the intended behavior matches the mechanism provided by the language.

GC-05-3 No Out-of-Bounds Access

When accessing an array or other data type, a program shall never read, write, execute or otherwise access outside of that data type, except when using language specific features that automatically allocate and, if necessary, initialize memory on demand. Note that, as described in the previous rule, there are language-specific mechanisms in Perl, Python and C++ that will automatically initialize data and render the requirement for explicit initialization irrelevant.

Examples:
If a Fortran array is indexed from 1 to 30, the code shall never read or write element 0 of that array because it is outside the bounds of the array. The code would be reading unrelated data, or data outside the process's memory pages.

If using a Perl hash, accessing element 0 is allowed because Perl will automatically create element 0, with value **`undef`**.

GC-06 Declarations

GC-06-1 Declare Variables Before Use

Codes shall declare all variables in languages where it is possible to do so. Later chapters clarify these rules on a per-language basis.

GC-06-2 Declare Types in Typed Languages

Codes shall explicitly define each type in typed languages that allow type definition. This rule does not apply to languages which are designed to be typeless. This rule is clarified in later chapters on a per-language basis.

Examples based on rules in later chapters:

- In Fortran this requires the use of **`implicit none`**.
- In C and C++ it requires declaring a function before the first use.
- This rule does not require type definitions in Python, which has no concept of a variable's type.

GC-07 Style Conventions

This section sets standards for coding styles. Here, coding style rules refer to rules where there are more than one correct way to do something, but one is chosen for the sake of consistency.

Justification:

Consistent coding style leads to more readable code.

GC-07-1 Consistent Style within Each Codebase

Coding style shall not vary within a codebase. Individual projects are allowed to define a coding style so long as it does not conflict with elements of coding style defined elsewhere in this document, and are allowed to define the scope of a "codebase" in which such a style is to be followed. If a project makes rules about coding style, developers shall follow those project's rules.

Justification:

Consistent coding style improves readability, but gives individual projects control over what coding styles are enforced, as described in GC-07-2. There are many coding styles; projects and users each have their own preferences. There are advantages and drawbacks to such things as variable name length requirements, spacing requirements and other arbitrary rules. However, within each codebase, the rules do not vary.

Example:
A developer editing a program that uses CamelCase variable names and uses four spaces per indentation level should continue to use the code base's conventions, or modify the code base's rules for a new convention. Note that a project may decide the scope of a "codebase" which could be as narrow as a portion of a file, or as broad as the entire project.

GC-07-2 Projects Specify Most Style Guidelines

The definition of what constitutes a "codebase" in GC-07-1 and the details of the style to be followed are left to project developers to decide, so long as they do not conflict with other requirements. *Such mandatory rules are defined elsewhere in this document, and may vary on a per-language basis.* Whatever style guidelines the project chooses, and those defined in this document, shall be followed by all developers of that project. It is reasonable to assume that in most cases a codebase would be defined as the contents of a source subdirectory, for example, everything in **`anything.fd`** or in **`likewise.cd`**.

GC-07-3 Nested Scopes Shall Be Indented

In a nested scope, such as a **`while`** loop, a **`subroutine`**, a **`class`**, or other nested block or definition, the code within the inner scope shall be indented further than the line at the top of the scope. In languages where one must end a scope with a line, that line shall be indented the same amount as the line at the top of the scope.

Example:
No indentation:

```
DO i=1,10
DO j=1,20
output_array(i,j)=input_array(i,j)**3+8
END DO
END DO
```

Bad indentation:

```
DO i=1,10
  DO j=1,20
    output_array(i,j)=input_array(i,j)**3+8
    END DO
    END DO
```

Good indentation:

```
DO i=1,10
  DO j=1,20
    output_array(i,j)=input_array(i,j)**3+8
  END DO
END DO
```

GC-08 Process Environment

GC-08-1 User and Local Paths Are Last in Path Variables

User-specific or relative directories shall occur last in path list variables.

There are environment variables used by the operating system or interpreters to find programs or libraries. Examples are **`$PATH`**, **`$PERL5LIB`**, and **`$PYTHONPATH`**.

Justification:
When a unix process runs a program without specifying its full path, the kernel searches the **`$PATH`** variable to find the program. If relative or user-specific directories are near the beginning of the **`$PATH`**, then the kernel may choose the wrong program or interpreter.

In particular, private "bin" directories should not be placed at the beginning of **`$PATH`** because they cannot be used in a production job. Personal directories should only be used if necessary to run an alternate version of a command or program normally run from a location in the default **`$PATH`**. Even in such situations, it is preferable to use modulefiles rather than setting **`$PATH`** directly.

Examples:
Bad:

```
export PATH=.:$PATH
setenv PATH $HOME/bin:$PATH
export RUBYPATH=$HOME/fancyrubyscripts:$RUBYPATH
```

Good:

```
export PATH=$PATH:.   # . is at the end of the $PATH
setenv PATH $PATH:$HOME/bin # $HOME/bin is at the end of $PATH
export RUBYPATH=$RUBYPATH:$HOME/fancyrubyscripts
```

SC: Scripting Language Standards

This chapter applies to languages for which the source code is interpreted rather than compiled to a native instruction set executable. Note that bytecode languages like Python do fall under this category because one executes the source code in a *.py file rather than *.pyc file.
Examples:

- Perl
- Python
- The “sh” family of shells
- The “csh” family of shells.

SC-01 Interpreter Specification

Programs that are not compiled and linked shall specify an interpreter in order for the kernel to know how to use them. This section places restrictions on interpreter specification.

SC-01-1 Shebang (#!) Line Is Mandatory

The first line of any executed script shall contain this *shebang line*:

```
#! /usr/bin/env interpreter
```

where *interpreter* is the name of the interpreter to use (ie.: bash, ksh, perl, python, …) The shebang (#!) must be the first two characters in the file. Note that this rule only applies to files executed as scripts. Files that are sourced (bash/sh/ksh/csh/tcsh) or included (Perl/Python) are never executed as a script, and hence have no need for a shebang line.

The only exception to this rule is POSIX sh, as discussed in SC-01-2

Justification and Clarification:

Why the Shebang?

The operating system kernel does not know how to execute a file unless:

1. the file is a compiled program, or
2. the file has a shebang line.

A script without a shebang line cannot be executed by the exec family of POSIX routines, nor by anything that relies on them such as Python’s os and subprocess modules. Some shells will still execute the program. They do this by detecting the kernel’s refusal to execute the program, and then interpret the file using a forked shell process instead. However, this behavior is not

well-defined (it is uncertain whether the shell will interpret the file), nor is it available in most languages.

Why /usr/bin/env?

The /usr/bin/env will search the **`$PATH`** to find the interpreter. This allows different versions to be maintained for backward compatibility. For example, ksh 88 vs. ksh 93 vs. AIX ksh, or Python 2.6.6 vs. Python 2.7.12.

SC-01-2: POSIX sh Scripts Shall Begin with #! /bin/sh

POSIX sh scripts shall begin with this line:

```
#! /bin/sh
```

Justification:
As described in the POSIX standard [ieee-posix], POSIX sh is available as /bin/sh in all POSIX-compliant operating systems.

SC-02 Related Scripting Standards

In some situations, there are related standards in NCEP that must be followed when writing scripts. This section refers to those standards and explains the situation in which they must be followed.

SC-02-1 Follow NCEP Implementation Standards

All scripts intended for implementation in the NCEP production suite shall follow the latest NCEP Central Operations - WCOSS Implementation Standards [ncepimpl].

SC-02-2 Environmental Equivalence Standards

All scripts intended for testing upgrades to the NCEP production suite shall follow the latest NCEP Environmental Equivalence standards.

CX: C and C++ Standards

C and C++ programs shall follow all standards described in the General Coding Standards guide as well as additional standards discussed in this section.

CX-01 Language

CX-01-1 Allowed C Language Versions

C code shall be written to conform to one of the following C standards. It is never acceptable to target an alternative, proprietary C standard.

- “C89” - ANSI X3.159-1989 "Programming Language C" and clarifications in ISO/IEC 9899/AMD1:1995
- “C99” - ISO/IEC 9899:1999
    - Standardizes many critical features, improves C++ compatibility, adds bool, static declarations, and many other critical features.
- “C11” - ISO/IEC 9899:2011

Although C89 is still allowed, code should migrate towards C99 and/or C11 as time permits due to improved features and portability of later C releases.

CX-01-2 Allowed C++ Language Versions

C++ code shall be written to conform to one of the following C++ standards. It is never acceptable to target an alternative, proprietary C++ standard.

- “C++98” - ISO/IEC 14882:1998
- “C++03” - ISO/IEC 14882:2003
    - Note that this fixes issues in the 1998 version, so the 2003 version is strongly recommended over the 1998 version.
- “C++11” - ISO/IEC 14882:2011
- “C++14” - ISO/IEC 14882:2014
    - Note that this fixes issues in the 2011 version, so the 2014 version is strongly recommended over the 2011 version..

CX-01-3 Exemption for Automatically-Generated Code

When C code is generated by an automatic code generator as an intermediate step in the compilation of source code in another language, that C code is exempted from all CX standards except those in the CX-01 section.

Justification:
The reason for this is that automatically-generated C code is not meant to be read by humans; the source that is converted to C is the actual source code.

Example:
Yacc (GNU implementation is Bison), which generates C code from Yacc code in a *.y file.

CX-02 Naming

CX-02-1 Filename Extensions

C and C++ files shall have the following filename extensions:

- *.c - C code
- *.cc or *.cpp - C++ code
- *.h, *.hh or *.hpp - C or C++ headers

CX-02-2 Preprocessor Symbols Cannot Begin or End with Underscore

Preprocessor symbols shall not be defined with names that begin or end with an underscore.

Justification:
As described in C and C++ standards, such symbols are reserved for compiler writers in later versions of C/C++ standards.

Example:
Bad:

```
#define _LINUX 1
```

Good

```
#define DETECTED_LINUX 1
```

CX-02-3 Avoid Common Names in Preprocessor Symbols

Code shall include a codebase-specific or situation-specific namespace indicator in symbol names. Ideally, this should be prepended to the symbol name, but it is allowed to have the namespace indicator elsewhere (such as a suffix or in the middle of the symbol).

Justification:
This is to avoid namespace clashes between preprocessor symbols. Preprocessor symbols are in a single namespace, and must share that namespace with the compiler and various libraries. For this reason, common symbols like **`LINUX`** or **`BIGENDIAN`** should be avoided. Instead, a name such as the library or program name should be prepended (**`G2_LINUX`**) or a name unlikely to have been chosen for another purpose should be used

Examples:
Bad:

```
#ifdef LINUX
do_something()
#endif
```

Good:

```
#ifdef G2_LINUX
do_something()
#endif
```

CX-03 Declarations

CX-03-1 Required C++ Class Contents

C++ classes shall contain a public default constructor, assignment operator, and copy constructor, or explicitly state in comments that they are using the compiler-generated version. If the class is expected to have subclasses, it shall have a virtual destructor.

CX-03-2 Avoid Overloading Operators for Non-Standard Purposes

Operators shall not be overloaded for non-standard purposes.

Justification:
C++ allows one to overload the meaning of operators to have meanings other than their intended mathematical, logical or I/O purposes. This can be abused in ways that make code highly counterintuitive.

CX-03-3 Use const Whenever Possible

The const keyword shall be used whenever possible.

Examples:

- If an argument is not modified, the argument should be const.
- If a member variable won't be modified after construction, it should be const.
- If an int pointer will not be incremented but its contents will be modified, the pointer (but not the target) should be made constant via `int const *`. This prevents accidental changes including changes caused by unintentionally passing an argument to a function that will modify it.

CX-04 Length

CX-04-1 Short ?: Blocks with Parenthesized Conditional

Any ?: expressions shall be less than 80 characters long, and the conditional shall be within parentheses. Note that if a `?:` expression is longer, it can be replaced with an if/else block or a function.

Justification:
The C/C++ `?:` block leads to inherently unreadable code, and should be avoided. However, it can dramatically simplify some code blocks.

Example:
Bad:

```
hemisphere = latitude>0?'N':'S';
```

Good:

```
hemisphere = (latitude>0) ? 'N' : 'S';
```

CX-05 Scoping

CX-05-1 Don't Use a Namespace in a Header File Global Scope

Code shall never put a **`using namespace`** statement at the global scope of a header file.

Justification:
This will clutter the global namespaces of all files that include the header file, which can cause bugs.

CX-05-2 Put Header-Accessible Symbols in Namespaces When Possible

All code that is available via a header file shall be placed in a namespace declaration if possible.

Justification:
This avoids clashes with similar names in other packages.

CX-05-3 Functions Should Be Reentrant

Functions that may be called multiple times in parallel (such as via threads or signal handlers) shall be reentrant.

Justification:

Not doing so can cause bugs when the code is run multiple times in parallel.

Examples:
Variables declared static at the function scope prevent re-entrancy since their storage is reused by all executions of that function. The only situation where this is acceptable is when the function will never be called by multiple threads or by a signal handler.
Bad:

```
int myfunction() {
    static int buffer[BUFFER_SIZE];
    ...do things...;
}
```

Good:

```
int myfunction() {
    int *buffer;
    buffer=malloc(sizeof(int)*BUFFER_SIZE);
    // Or put it on the stack with int buffer[BUFFER_SIZE]
    ...do things...;
}
```

CX-06 Preprocessing

CX-06-1 Header Files Shall Always Have Header Guards

Code shall always use header guards in header files. A header guard is a special ifdef block that prevents a header file from being processed more than once.

Justification:
This prevents accidental redeclaration of symbols and infinite #include loops.

Example:
Bad version of myheader.h:

```
int myfunction();
```

Good version of myheader.h:

```
#if ! MYHEADER_H
#define MYHEADER_H
int myfunction();
#endif
```

CX-06-2 Don't Use Macros Unless Absolutely Necessary

Code shall never use macros unless absolutely necessary. Note that in C and C++, most possible uses of preprocessor macros are unneeded and should be avoided.

Justification:
Macros are unavailable to the debugger, making them harder to debug. Macros are typeless, leading to unexpected type conversion problems. Inline functions and constants are available to the debugger, and are just as fast at runtime.

Examples:
Bad:

```
#define MAX(I,J)    ( (I>J) ? I : J )
```

Good alternatives:

```
// C++ template inline function style (any type):
template<class T>
inline const T&max(const T&left, const T&right) {
    return (left>right) ? left : right;
}

// C inline function style:
static inline int max(int I,int J) {
    return (I>J) ? I:J;
}
```

Bad:

```
#define TWO 2
```

Good alternatives:

```
// C++ template class style (any type):
template<class T>
class myconstants {
public:
    static const T TWO=2;
};

// C style:
static const int TWO=2;
```

CX-06-3 Use #if instead of #ifdef for Option Specification

Code shall use #if instead of #ifdef when enabling or disabling sections of code via options.

Justification:
This avoids the situation where someone may attempt to disable the **`#ifdef`** option via **`-DOPTION=0`**

Example:

Bad:

```
my_c_code.c:
    #ifdef MY_OPTION
    fancy_c_code;
    #endif
user@machine> cc -DMY_OPTION=0 my_c_code.c
```

In this example, fancy_c_code was enabled due to the `#if`. The user expected that it would be disabled by setting the `MY_OPTION` flag to `0`. The safe alternative is:

Good:

```
my_c_code.c:
    #if MY_OPTION
    fancy_c_code;
    #endif
user@machine> cc -DMY_OPTION=0 my_c_code.c
```

Now fancy_c_code is disabled, as the user intended.

CX-06-4 C Interfaces Shall Use extern "C" Guards

C headers that may be used by C++ shall have extern "C" guards. Note that extern "C" can be declared on a per-symbol basis. That is acceptable as well.

Example:

Bad:

```
void my_fancy_C_function(void);
...more declarations...
```

Good example:

```
#ifdef __cplusplus
extern "C" {
#endif
void my_fancy_C_function(void);
...more declarations...
#ifdef __cplusplus
} // End extern "C"
#endif
```

CX-06-5 No Data Definitions in Header Files

Header files shall never contain data definitions unless they are const or static.

Example:

Bad header:

```
int myvar=5;
```

Alternatives:

```
static const int myvar=5;
#define MYVAR 5
```

Or place `int myvar=5` in a *.c or *.cc file.

CX-07 The Goto Statement

CX-07-1 GOTO Only Allowed in Certain Circumstances

The `goto` statement shall not be used unless explicitly allowed by later rules in the CX-07 section.

Justification:
In nearly any situation where a **`goto`** is used, it is possible to replace it with something more readable and maintainable.

CX-07-2 GOTO Allowed for Error Handling in C at End of Function

In C, but NOT C++, it is acceptable to use **`goto`** to jump to an error handling block at the end of a routine before returning or stopping. It is *never* acceptable to use **`goto`** for clean-up blocks in C++. Exceptions or destructors shall be used instead.

Note that, even in that situation, several **`IF`** blocks or clean-up functions are likely to be clearer ways of expressing the code. Such approaches should be seriously considered before using a **`GOTO`**.

Example:
An allowed “clean-up” goto in C:

```
    my_files=open_my_files()
    if(!my_files) goto cleanup
    big_array=(float*)malloc(sizeof(float)*big_array_size)
    if(!bigarray) goto cleanup
    // ... lots of code here to work magic on my_files
    //     and big_array ...
cleanup:
    if(my_files)
        close_my_files(my_files);
    if(big_array)
        free(big_array);
```

CX-07-3 Goto Allowed for Automatically-Generated State Machines

It is acceptable to use **`goto`** to implement automatically-generated state machines. However, when possible, a loop should be used instead of a case statement.

Justification
This is because the internal logic of a state machine is most clearly and efficiently represented by **`goto`**.

CX-07-4 Goto Allowed for Exiting or Continuing Outer Loop

It is acceptable to use a **`goto`** to continue to the next cycle of an outer loop or to exit that loop entirely. In such situations, the goto shall be placed at the end of the loop (just before the "`}`") or just after the loop.

Note that one can frequently achieve better readability without a **`goto`** through a return statement or by using flags to skip inner loop iterations.

FT Fortran Standards

This chapter discusses additional requirements for Fortran code. All Fortran programs are also required to follow rules discussed in previous sections, with the exception of the SC Scripting Language Standards section. Most of these requirements are from Eugene Mirvis's research in [mirvis2016s] and [mirvis2016g], and a few are from discussions whose results are summarized in [coding2016c].

FT-01 Language

FT-01-1 Allowed Language Versions

Code shall be written towards a targeted Fortran version which shall be one of the below listed ISO/IEC international standard versions of Fortran. It is never acceptable to target a specific compiler's version of Fortran, such as targeting the gfortran subset of Fortran 2003.

Note that later rules add exceptions, such as allowing the Fortran 2008 new unit argument to the OPEN statement.

The only acceptable target language versions are:

- Fortran 90: ISO/IEC 1539:1991
- Fortran 95
    - Base language: ISO/IEC 1539-1:1997
    - Variable length character strings: ISO/IEC 1539-2:2000
    - Conditional compilation: ISO/IEC 1539-3:1998
- Fortran 2003
    - Base language: ISO/IEC 1539-1:2004
    - Enhanced module facilities: ISO technical report *TR-19767:2005 Enhanced module facilities in Fortran*

Justification:
Code written for proprietary Fortran versions reduces portability.

Why not Fortran 2008?
Fortran 2008 (ISO/IEC 1539-1:2010) is not widely supported yet. A Fortran community webpage tracking the status of Fortran 2008 support can be found here:
http://fortranwiki.org/fortran/show/Fortran+2008+status
It is notable that, as of this writing, Cray has full Fortran 2008 support and several other compilers support most Fortran 2008 features.

However, several compilers have abandoned support for any Fortran release after Fortran 95 and hence have extremely incomplete support for Fortran 2008, and even Fortran 2003. It is unwise to continue using such compilers for operational work in the future as the lack of ability to support new standards is an indication of lack of maintenance of the compiler itself.

FT-01-2 The Ten Year Rule

Fortran standards whose publication date is less than 10 years old shall not be used. Note that the publication date is always later than the title date; Fortran 2003 was published in 2004 and Fortran 2008 in 2010. The latest such standard as of this writing is Fortran 2003. Special exceptions are made for certain features of newer standards versions.

Developers can use any feature of any Fortran standard published at least ten years ago that is not yet banned by this document. Special exceptions are made elsewhere in this document for certain language features in newer standards.

Justification:
It takes time for compiler developers to adopt new standards. However, if a compiler is unable to adopt standards that are a decade old, that is an indication that the compiler is no longer suitably maintained or is still being developed towards usefulness. Such a compiler should not be used for operational work nor should its limitations be imposed as a restriction of operational work.

FT-01-3 C Preprocessor Lines

All C preprocessor lines in Fortran code shall follow all C standards in sections CX-01, CX-02, CX-04 and CX-06.

It is acceptable, but discouraged, to use C preprocessors to preprocess Fortran source code. Any C preprocessing lines are C lines and shall follow all C/C++ standards in sections CX-01, CX-02, CX-04 and CX-06.

Clarification:
Among other things, this means that all Fortran headers must have header guards. They have file length limitations and line limitations that may restrict further than what is listed here. All comments on those lines must be valid C comments. Preprocessor symbols are required to not begin or end with underscore (_) and are required to follow naming conventions described in the CX chapter.

Justification:

Note that C preprocessing is NOT part of the Fortran language; any C preprocessing lines are C lines. Those are lines beginning with a hash mark (**#**) Hence, it is acceptable, but discouraged, to use C preprocessors to preprocess Fortran source code.

Example:
Bad:

```
#if .not. OPTION   ! This line is a syntax error in C
this_is_fortran="code"
#endif ! Invalid comment syntax
```

Good:

```
#if ! OPTION   /* This is a line of C code */
this_is_fortran="code"
#endif /* Valid comment syntax */
```

FT-01-4 No Fixed-Form Fortran

Fortran code shall be written in free form; when an exception is granted for the continued use of fixed form, the filename extension shall follow rule FT-04-1.

FT-01-5 Fortran 2008 OPEN Statement "newunit" Argument is Allowed

The **`newunit`** argument to the OPEN statement is part of Fortran 2008, but it is allowed in NCEP codes. This is a special case exemption from rules FT-01-1 and FT-01-2.

Justification:
This solves a major, long-standing problem in Fortran I/O: there has been no easy way to know which unit numbers are available for file I/O. If a unit number choice is hardwired, it may conflict with earlier or later code that uses that unit. This has led projects to assign unit numbers or ranges in the design phase of the program. However, that does not work with in programs such as NEMS or WRF where the number of files to open is quite large.

A workaround frequently used is to designate a range of unit numbers (say, 1000-2000) as temporary unit numbers. The program then uses a loop with an INQUIRE statement to search for an available unit number. This causes two problems. First, it may collide with a pre-designated unit number used by a library after the program is already using the number. For example, an I/O library wants to use **`1010`** for reading a configuration file, but the program has already opened **`1010`** due to its automatic loop. Secondly, it requires a potentially costly loop that may run out of unit numbers.

Example:

```
integer :: unit
open(file="inputfile",newunit=unit,... more arguments to open …)
read(unit) bigarray
```

FT-01-6 Maximum of 132 Characters Per Line

Lines shall never exceed 132 characters, including lines produced by pre-processors such as the C Preprocessor.

Justification:
The Fortran 2003 standard dictates a limit of 132 characters per line. Some compilers may support longer lines, but they would make code compiler-specific and reduce portability. Note that longer lines can be split up using line continuation characters (`&`). The Fortran 2003 standard allows a limit of 255 lines in a sequence of continuations.

FT-02 Declarations

FT-02-1 Implicit None

All blocks of Fortran code that have dummy arguments, use modules, or have variables shall contain the "implicit none" declaration to ensure that types and variables are explicitly declared.

FT-02-2 Block Order

Subparts of a fortran block shall be in this order. However, items #1 and #2 may be reversed, if necessary for the documentation generation system or project style requirements.

1. Block documentation
2. Block declaration: **`program`**, **`function`**, **`module`**, **`subroutine`**.
3. Any **`use`** statements
4. **`implicit none`**
5. **`private`**
6. **`public`** declarations on a per-variable basis
7. If relevant, dummy arguments
8. Locals and parameters
9. Source code.
10. The **`contains`** section.

FT-02-3 One Declaration Per Line

Only one variable may be declared per line of code, unless the variables on the line are all closely related.

Example:
Bad:

```
integer :: temperature(nx,ny), domains(ndom)
```

Good:

```
integer :: temperature(nx,ny)
integer :: domains(ndom)
```

FT-02-4 Save Variables Shall Be Declared As Saved

Code shall explicitly declare a save variable to be a save variable.

Justification:
This requirement is to ensure that users catch unintentionally saved variables which can lead to common bugs. For example, the second call to the subprogram may have different results due to save variables having different values. Threading or signal handling will result in unpredictable results due to lack of re-entrancy.

Example:
Bad:

```
real function mfc100(angle)
   real :: angle
   real :: conversion=50/3.14159
   ... do things ...
end function mfc100
```

Good:

```
real function mfc100(angle)
   real :: angle
   real, save :: conversion=50/3.14159
   ... do things ...
end function mfc100
```

FT-02-5 Constants Shall Be Parameters

Compile-time constant values shall be declared as parameters.

Example:
Bad:

```
real :: e
e=2.71828182846
if(signature=="GRIB") then
```

Good:

```
real, parameter :: e=2.71828182846
character(len=4), parameter :: GRIB_signature="GRIB"
if(signature==GRIB_signature) then
```

FT-03 Datatypes

FT-03-1 Use Logical Type for Logical Variables

If a variable is intended to hold a true/false value, it shall be declared as a logical, unless necessary for inter-language interaction, communication, interacting with other code bases, or I/O.

Examples of acceptable situations where logical should not be used:

- Passing **`iso_c_binding`**'s **`C_BOOL`** to C is necessary for C interaction.
- Converting to or from another representation in a file in a codec for data storage and retrieval.

FT-03-2 Use iso_c_binding and Bind(C) for C Interaction

Fortran code that interacts with C shall use **`bind(C)`** and the **`iso_c_binding`** module to do so. Any C functions used in Fortran shall have Fortran declarations with **`bind(C)`** and datatypes from iso_c_binding. Any Fortran functions called from C shall use **`bind(C)`** and have all arguments and return values declared using **`iso_c_binding`**.

Justification:
This ensures cross-platform compatibility without the need for guessing C-Fortran interaction data types and name mangling schemes as was necessary in Fortran 95 and earlier.

FT-04 Naming

FT-04-1 Filename Extensions

Fortran files shall follow these naming conventions:

- **`filename.f`** - Fixed-form fortran file.
- **`filename.f90`** - Free-form fortran file, which may contain code from a later version than Fortran 90
- **`filename.F`** - Input to a pre-processor to create a filename.f
- **`filename.F90`** - Input to a pre-processor to create filename.f90
- **`filename.h`** or **`filename.inc`** - input file to be **`#include`**d or **`include`**d in another Fortran source

Fixed-Form Note: Although we specify the file extension for fixed-form Fortran, fixed-form Fortran is not actually allowed according to rule FT-01-4. Hence, these file extensions are only for code that was granted an exemption.

Lower-case extension Fortran code (*.f, *.f90, etc.) shall never be pre-processed. They are direct input to the compiler.

FT-04-2 Use Named End Statements

Named blocks shall contain their name in the end statement. If a conditional or looping construct is longer than 50 lines or contains more than five sub-blocks, it shall be named.

Justification:
This is an important safeguard in Fortran that prevents accidentally ending the wrong block.

Bad:

```
subroutine refactorme(chickens)
  if(chickens>100) then
    ! Do something with more than 100 chickens
    ...many lines of code here...
  endif
end
```

Good:

```
subroutine refactorme(chickens)
  bigif: if(chickens>100) then
    ! Do something with more than 100 chickens
    ...many lines of code here...
  endif bigif
end subroutine refactorme
```

FT-05 Scoping

FT-05-1 Module Private By Default

Module and class variables shall be declared **`private`** by default. Public symbols shall be individually be declared **`public`**.

Example:
Bad:

```
module physics
  implicit none
  contains
  subroutine microphysics...
  subroutine micro_helper...
```

```
    ... more subroutines
end module physics
```

Good:

```
module physics
  implicit none
  private
  public :: microphysics
  contains
  subroutine microphysics...
  subroutine micro_helper...
  ... more subroutines
end module physics
```

FT-05-2 Private Member Variables

In Fortran classes, member variables shall all be **`private`** and have accessor and mutator routines. The purpose of this rule is to allow subclasses to provide different functionality. Note that this does *not* apply to simple types that are used for data storage; it only applies to classes.

In some rare cases, it may be necessary for speed to provide an informational public member variable to avoid the double dereference and call stack necessary for a virtual function call. In such situations, the **`public`** member variable should be an informational copy of another variable used internally, and should be updated as needed only by the class itself.

Example:

Bad:

```
type, extends(GRIB2Record) :: SyntheticSatellite
    integer :: constellation
    ...more data...
end type SyntheticSatellite
...
print *,myrecord%constellation
```

Good:

```
type, extends(GRIB2Record) :: SyntheticSatellite
    integer, private :: constellation
    ...more data...
contains
    module procedure getConstellation
    module procedure setConstellation
end type SyntheticSatellite
...
print *,myrecord%getConstellation()
```

FT-06 Obsolete Features

Regardless of which targeted Fortran standard is chosen (see FT-01), the obsolete features of Fortran listed in this section shall not be used.

FT-06-1 Never Use Arithmetic If

The Arithmetic If was declared obsolescent in Fortran 90 and shall not be used. Note that it can be replaced with an **`if-elseif-else`** block or case statement.

Example:

```
IF (numeric_expression) negative,zero,positive
```

FT-06-2 Never Use Assigned Goto

The Assigned **`Goto`** was declared obsolescent in Fortran 90 and shall not be used. Note that it can be replaced with a case statement or **`if-elseif-else`** block.

Example:

```
assign 10 to i
goto i
```

FT-06-3 Only Goto a Continue

When a **`GOTO`** statement is used, it shall only jump to a continue statement.

Justification:
This is to prevent common programming bugs.

Example:
Bad:

```
      goto 30
      ...code here…
30 print *,result ! Bad!
```

Good:

```
      goto 30
      ...code here…
30 continue ! Good
      print *,result
```

FT-06-4 No DO (NUMBER) Loops

Obsolescent **`do`** loop style of **`do 10...10 continue`** shall not be used.

Justification:
This is to prevent common programming bugs.

Example:
Bad:

```
         do 10, i=1,ni
            data(i)=some calculation
      10 continue
```

Good:

```
      do i=1,ni
         data(i)=some calculation
      end do
```

FT-06-5 No Pause Statements

The **pause** statement is an obsolescent feature that pauses execution and waits for input. This feature shall not be used. Note that this can be replaced by a **write** statement and a **read** statement to retain compatibility with later Fortran standards.

Justification:
This feature was deprecated in Fortran 90 and removed from Fortran 95.

FT-06-6 DO Loop Counters Shall Be Integers

DO loop counters shall be **integer** valued, not **real** valued.

Justification:
Numeric imprecision of **real** calculations lead to unpredictable behavior of loops.

Example:
Bad:

```
      real :: r
      do r=1.0,10.0
         ...stuff happens...
      enddo
```

Good:

```
      integer :: i
      do i=1,10
         ...stuff happens...
      enddo
```

FT-07 The GOTO Statement

FT-07-1 GOTO Only Allowed in Certain Circumstances

The **GOTO** statement is shall not be used except for certain situations.

Examples:

- Exiting a loop shall be done with the **exit** statement.
- Skipping to the next iteration of a loop shall use the **cycle** statement.
- One can **exit** or **cycle** a loop at any nesting depth.
- **If(expr)goto** can be replaced with a loop or function call.

FT-07-2 GOTO Allowed for Error Handling at End of Subprogram or After a Loop

The only acceptable use of **GOTO** in hand-coded Fortran is to jump to an error handling block at the end of a routine or just after a loop. Note that, even in that situation, several **IF** blocks, a contained subroutine, or destructors are likely to be more clear ways of expressing the code. Such approaches should be seriously considered before using a **GOTO**.

Example of allowed "clean-up" GOTO:

```
      call myFiles%open(ierr)
      if(ierr/=0) goto 20
      allocate(someReallyBigArray(myFiles%xspan,myFiles%yspan),
               stat=allocerr)
      if(allocerr/=0) goto 20
      ! ... code that uses myFiles and someReallyBigArray ...
20    continue ! Clean-up section
      if(allocated(someReallyBigArray)) &
           deallocate(someReallyBigArray)
      if(myFiles%isopen()) &
           call myFiles%close()
```

FT-07-3 GOTO Allowed for Automatically-Generated State Machines

It is acceptable to use **GOTO** to implement automatically-generated state machines. Note that a loop can be used instead of a **case** statement.

Justification:
The internal logic of a state machine is most clearly and efficiently represented by **GOTO**.

MK: Makefile Standards

These rules apply to makefiles. Most of the rules are inspired by, but not copied from, the GNU Style Guide [gnu-style] Makefile chapter. Note that makefiles shall also follow the GC: General Coding Standards rules and SG: Standards Governance rules. Makefiles do not need to follow the SC: Scripting Standards or the specific language standards; they are not considered to be scripts or compiled code.

MK-01 Variables

MK-01-1 Specify Shell to Use for Build Commands

All makefiles shall contain the following line. It is acceptable to obtain this line via a make include file instead:

```
SHELL=/bin/sh
```

Justification:
On some platforms, and in older versions of GNU Make, the shell may be the user's login shell or some other incorrect default.

MK-01-2 Use Variables for Build Targets

The build targets shall be configurable via variables.

Example:
Bad:

```
MPICC=mpicc
myprog: myprog.c
	$(MPICC) -o myprog myprog.c
```

Good:

```
MPICC=mpicc
MYPROG=myprog
$(MYPROG): myprog.c
	$(MPICC) -o $(MYPROG) myprog.c
```

MK-01-3 Use Variables for Directories

All directories shall be specified via make variables.

Example:

Bad:

```
MPICC=mpicc
MYPROG=myprog
myprog: myprog.c
	$(MPICC) -I /path/to/bzip/include -o myprog myprog.c
```

Good:

```
MPICC=mpicc
MYPROG=myprog
BZIP_INC=/path/to/bzip/include
myprog: myprog.c
	$(MPICC) -I $(BZIP_INC) -o myprog myprog.c
```

MK-02 Utility Executables

MK-02-1 Only Use Standard Executables

Only the following executables may be used by the makefile directly. Any other executable shall be configurable via a script or other mechanism before running make:

awk cat cmp cp diff echo egrep expr false grep install-info ln ls mkdir mv printf pwd rm rmdir sed sleep sort tar test touch tr true ar bison cc flex install ld ldconfig lex make makeinfo ranlib texi2dvi yacc

MK-02-2 Use Variables For Executables

All executables used by make rules shall be used via a variable.

Example:

Bad:

```
$(MYPROG): myprog.c
	mpicc -o $(MYPROG) myprog.c
```

Good:

```
MPICC=mpicc
$(MYPROG): myprog.c
	$(MPICC) -o $(MYPROG) myprog.c
```

MK-03: Build Rules

MK-03-1 Specify All Suffixes

All suffixes used in build rules shall be specified by the **`SUFFIXES:`** … variable.

MK-03-2 Required Targets

All makefiles shall specify the following targets, and the **`all`** target shall be the default target:

- **`all`** - build all targets, but do NOT install in final locations.
- **`install`** - build all targets AND install in final locations.
- **`clean`** - deletes only files that are normally not in the development repository but are created by the build process.
- **`uninstall`** - deletes files that would be installed by **`install`**.
- **`test`** - run tests.

It is acceptable for the **`test`** target to do nothing if no tests exist. However, it must still exist and succeed.

PL: Perl Standards

All perl scripts and modules shall follow the requirements in the *GC: General Coding* and *SC: Scripting Languages* section in addition to requirements in this section.

Perl 5 should be phased out and replaced with alternatives, such as shell, Python or compiled languages. However, there are situations where Perl has to be used due to its excellence as a text parsing language. There are alternative languages that would fill this niche, and those languages should be evaluated for future use in operations.

We suggest limiting Perl 5 usage in operations to complex text parsing tasks, and eliminate all other usage as time permits. NCEP should investigate alternative languages to fill Perl 5's important niche in operations.

Justification:
It is the judgement of the Coding Standards Group that Perl 5 is designed in such a way that it encourages bad programming practices. Furthermore, most uses of Perl 5 in the NCEP production suite could be better represented in ksh93, bash or Python. Hence, the use of Perl 5 should be phased out in the production suite with one exception.

Perl 5 satisfies one critical niche in the NCEP production suite: complex text parsing tasks. There is no other operationally-approved language suitable for this purpose. The CPython 2.6.6 implementation of Python is extremely slow at text parsing (nominally 10x slower than Perl 5 on WCOSS). C, C++ and Fortran have no native regular expression support. Bash and ksh93 have native language regular expression support, but tend to be far slower even than Python at that task. There are other languages suitable for this purpose that are installed on the operational supercomputer, notably Ruby, but they are not yet approved for operations.

For this reason, we suggest limiting Perl 5 usage in the production suite to complex text parsing tasks, and eliminate other usage as time permits. That is not a requirement in this document; it is just a strong recommendation.

Furthermore, we suggest investigating alternative languages to fill this niche in operations.

PL-01 Language Version

PL-01-1 Allowed Versions

Perl code shall be written for a version of Perl that is at least 5.4, but earlier than 6.

Justification:
Perl 6 is not yet supported at NOAA, and earlier versions than 5.4 are not allowed due to lack of some critical language features.

PL-01-2 Follow the Perl Style Guide

Every version of Perl comes packaged with a style guide ([perlstyle]). Perl scripts and modules shall follow the specific rules in that guide.

PL-02 Dangerous Features

PL-02-1 No Punctuation Character Variable Names Except $_, @_, $?, $!, $|, $$, and $@

There are built-in Perl variables whose names consist solely of punctuation characters. These shall not be used, with the exception of **`$_`**, **`@_`**, **`$?`**, **`$!`**, **`$|`**, **`$$`**, and **`$@`** as those have widespread known meanings.

Note that there are english language versions of the punctuation character variable names accessible via the built-in “English” perl module.

PL-02-2 Always “use strict”

All perl scripts and modules shall begin with **`use strict`** near or at the top of the script

Justification:
“Use strict” catches many common programming problems.

PL-02-3 Never Override Built-In Variables

Code shall never override the meanings of built-in Perl variables such as **`$a`** and **`$b`**.

Justification:
This is a common source of bugs in perl programs.

PL-03 Variables

PL-02-3 No $_ Except in Single-Line Code Blocks and Anonymous Code Blocks

Code shall not use the **`$_`** variable except in single-line Perl scripts or single-line anonymous code blocks.

Justification:
Use of **`$_`** reduces readability in a large code block. However, its use in single-line code where **`$_`** is implied simplifies code dramatically but does not reduce readability.

PL-02-4 Declare Variables

Code shall always declare variables before first use via **`my`**, **`our`**, or other related mechanisms.

PY: Python Standards

This chapter applies to all Python code, including modules, packages, scripts, direct execution of python -c, and embedded python code. All Python code shall also follow GC: General Coding Standards and SC: Scripting Standards rules. Note that many of these rules come from version 2.59 of the Google Python Style Guide, which does a good, critical, review of the Python language and the disadvantages of the more advanced features of that language.

PY-01 Language Version

PY-01-1 Language Versions

Python code shall target any one of a range of language versions. Any language version from 2.6.9 onward that is a release version is allowed. However, be aware that only 2.6 is installed on all NOAA machines at this time.

PY-01-2 RedHat Python 2.6.6 Allowed as a Special Case

It is acceptable to use RedHat Python 2.6.6 so long as the code does not rely on the few unpatched bugs in that version. However, it is not acceptable to use the stock Python 2.6.6 from the Python website.

Justification:
The stock Python 2.6.6 downloaded from the Python website has security vulnerabilities, so the stock version of Python 2.6.6 should never be used under any circumstances. The RedHat version has patched the vulnerabilities. This is the version present on most NOAA machines, except for WCOSS Cray, which has 2.6.9.

PY-01-3 No Deprecated or Broken Python Features

Any features of the Python language that are deprecated or non-functional in the targeted range of versions (see PY-01-1) shall not be used, unless the code provides multiple implementations or workarounds for multiple versions (such as to support Python 2 and 3).

Example:

- The subprocess module should not be used to launch multi-stage process pipelines if the package targets Python 2.6.6 due to known bugs in that version of Python.
- The **`string.atof`** should not be used since it is deprecated; as defined in the **`string.atof`** documentation, **`str.float()`** should be used instead.

PY-02 Style

PY-02-1 No Indentation Tabs

Python scripts shall never use tab characters for indentation. Note that this is a syntax error in Python 3.

Justification:
Python scoping is controlled by the number of spaces before the first non-space character in the line. Different editors interpret tabs in different ways, which has led to so many errors that Python 3.x considers tabs a syntax error. In Python 2, the presence of indentation tabs can be checked by using the -t option. One -t turns on warnings about tabs and two (-tt) will treat a tab as an error.

PY-02-2 Maximum of 80 Characters Per Line

Python lines shall not exceed 80 characters in length.

PY-02-3 One Statement Per Line

Semicolons (`;`) shall not be used to put multiple statements in one line.

PY-03 Scoping

PY-03-1 Import from Modules Only

Symbols shall only be imported from modules, specified from the outermost scope.

Example, the following is allowed:

```
import os.path # make os module visible as os.path
from os.path import isdir # make isdir locally visible as isdir
# Avoid clash between os.open with built-in open by
# importing os.open as "osopen"
from os import open as osopen
```

PY-03-2 Use Full Module Path

Never use relative paths to modules in an import. Instead, import statements shall be relative to the top of the module hierarchy.

Justification:

This avoids conflicts in module names.

PY-03-3 Avoid Global Variables

Do not use module-scope or package-scope variables, even in scripts. If global variables are absolutely needed, they can be made internal to a module.

PY-03-4 Nested Classes and Functions

Functions or classes can be declared inside other functions and classes.

Justification:
This can greatly simplify code and hide implementation from unintended usage. This is a good feature, and should be used.

PY-03-5 Lexical Scoping

A nested function, as described in PY-03-4, can refer to variables in the outer defining scope. This should be used for the same reasons as PY-03-4.

PY-04 Iteration

PY-04-1 List Comprehension

List comprehension shall be used where it makes the code simpler and easier to read.

Justification:
Complex list comprehension with multiple levels of comprehension is hard to follow and should be avoided.

Example:
Needlessly complicated way to express the Fibonacci numbers:

```
[1] + [reduce((lambda a,x:
    [1,1] if x<2 else a+[a[-1]+a[-2]]),
        range(n+2))[-1] for n in xrange(10)]
```

Good way that uses the golden ratio:

```
[round(( (1+sqrt(5)) /2 )**n/sqrt(5)) for n in xrange(11)]
```

Good way that uses a simple loop:

```
fib=[1,1]
for x in xrange(9):
  fib.append(fib[-2]+fib[-1])
```

PY-04-2 Use Default Iterators

Default iterators shall be used whenever possible, instead of looping over indices or other methods.

Justification:
Some types, like **`dict`**, **`list`** and **`file`**, provide default iterators which are simple and efficient. They iterate over a datatype without having to create intermediate objects or method calls.

Example:

```
with open("somefile","rt") as somefile:
    for line in file:
       do something with the line
```

PY-04-3 Use Generators as Needed

Generators are classes or functions like **`xrange`** that return an iterable object. This simplifies code.

Example:

```
for i in xrange(10): # loop from i=0 to i=9
    do something with i
```

PY-05 Expressions

PY-05-1 Conditional Expressions for Simple Expressions Only

The ternary conditional operator is allowed for simple expressions (shorter than 72 characters). Note that they can be replaced with an **`if`** block or function call instead.

Example:
These are constructs like:

```
hemisphere="S" if lat<0 else "N"
```

PY-05-2 Lambda Functions Shall Be Simple

`Lambda` functions are anonymous functions that can only consist of an expression. They shall never exceed one line. If they must exceed one line, a lexical scope function shall be used instead..

Justification:
They are harder to debug but can be convenient and reduce code length.

Example:

```
mylist.sort(cmp=lambda a,b: a.id<b.id)
```

PY-05-3 Use Implicit Boolean for Logical Evaluation

Implicit boolean is allowed, but shall be used only when an arbitrary, typeless, truth check is intended.

Justification:
Python objects have implicit boolean conversions which are generally faster and more readable than comparison operators. However, they must be used carefully. Keep in mind that the empty string is `False`, and so is `0`. If the intent is to check whether a variable is `None`, `is None` should be used.

Example:
Bad:

```
if some_dict=={}:
    do things
```

Good:

```
if not some_dict:
    do things
```

PY-06 Declarations

PY-06-1 Default Argument Values Shall Be Constant

Default values in function or method argument lists shall not be mutable objects.

Justification:
In functions with a long argument list, it is convenient to have default values so that not all arguments must be specified. However, keep in mind that default values are interpreted at module evaluation time. Hence, if the default value is a list, dict or other mutable object, the same object is reused for all calls. This can lead to counterintuitive bugs.

Example:
Bad:

```
def do_things(var1='a',var2=[]):
    do things here
```

Good:

```
def do_things(var1='a',var2=None):
    if var2 is None:
```

```
            var2=[]
        do things here
```

PY-06-2 Use Properties Instead of Light-Weight Getter/Setter Methods

Properties shall be used instead of simple getter/setter methods whenever possible, but they should be simple. Getter properties shall not modify the class except to set a cached value, if needed.

Example:
Bad (without properties):

```
class square(object):
    def __init__(self,length):
        self.__length=length
    def getarea(self):
        return self.__length**2
…
print square(8).getarea()
```

Good (with property)

```
class square(object):
    def __init__(self,length):
        self.__length=length
    @property
    def area(self):
        return self.__length**2
…
print square(8).area
```

PY-06-3 Decorators Shall Be Used Only When Needed

Decorators are allowed only when there is a clear advantage in their use. This rule does not apply to `@property`, which is covered by the previous rule.

Clarification:
Python allows method decorators, the most well-known of which are **`@classmethod`** and **`@staticmethod`**. However, users can define their own decorators. Decorators should be simple; no external dependencies beyond the Python interpreter itself. They should be used judiciously and only when there is a clear advantage.

PY-06-4 Classes Shall Derive From a Superclass

All classes shall derive from a superclass. Base classes should derive from object.

Justification:
This is needed to make several features of the python language work properly.

PY-07 Exceptions

PY-07-1 Exceptions Are for Error Handling

Python Exceptions shall be used only for error handling. There may be a few rare cases where Exceptions must be used for other purposes for efficiency reasons. Such cases shall be well-documented.

PY-07-2 Catch the Narrowest Exception Type Possible

When catching exceptions (**`try...except`** block) the code shall catch the narrowest type possible. It should never do **`except:`** or catch anything above **`Exception`** in the Python exception hierarchy.

PY-07-3 Use “finally” for Clean-Up Code

If a code needs to clean up some resources no matter what happens in a **`try...except`** block, it should use the **`finally`** construct instead of catch-all **`except:`** blocks.

PY-08 Unsafe Language Features

PY-08-1 Do Not Rely on Atomicity of Built-In Types

Class methods in built-in types like **`dict`** and **`list`** may not be atomic operations. If threading support is critical, use the **`threading`** module and its locking functionality.

PY-08-2 Do Not Use Power Features

Features such as metaclasses, bytecode access, on-the-fly compilation, dynamic inheritance, object reparenting, import hacks, reflection or modifying system internals.

Justification:
These features lead to unmaintainable code and can cause difficulty porting to other implementations of Python.

PY-08-3 Explicitly Close Files and Sockets

Codes shall explicitly close any open files and sockets as soon as they are no longer needed.

Justification:
Python garbage collection may take an unpredictable amount of time to automatically close file and socket objects. In some situations, they may stay open for the entire duration of the program, such as if there are cycles in the object reference graph. It is critical to explicitly close files and sockets or manage them using the **`with`** statement.

PY-08-4: Declare a Main Program Function

The main program shall never be at the script scope.

Justification:
This causes problems with some python scanners and linters which must import a script in order to scan it. Such tools will end up executing the script instead of merely scanning its contents.

Example:
Instead, use a **`main`** function and call it like so:

```
def main():
     Main program goes here.

if __name__ == '__main__':
   main()
```

SH: Shell Scripts

These standards apply to scripts written in POSIX sh, C shell (csh), tcsh, Korn Shell (ksh) and Bourne Again shell (bash), as well as other similar shells. Any such scripts shall also follow all GC: General Coding standards and SC: Scripting Standards. This section has very few rules because most possible rules are already covered by those sections.

SH-01 Languages

SH-01-1 List of Allowed Languages for Scripts

Scripts shall be POSIX sh (IEEE 1003.1), Bourne Again Shell (bash) or the Korn Shell (ksh), as those are widely-supported. Large shell projects (larger than 400 lines) should use Bourne Again Shell (bash) version 4 or later or the 1993 or later version or later of Korn Shell (ksh93). The C Shell (csh) and tcsh are not currently supported in NCEP production and therefore their use is discouraged.

This requirements document does not outright ban the use of csh or tcsh for automation, but we do suggest moving away from those shells to better-suited alternatives.

Justification:
The bash, ksh and sh scripting languages are encouraged over older alternatives because they add important features like type checking, improved support for functions, regular expressions, and additional data types. These new features can reduce code length and avoid bugs (ie.: local variables in functions).

SH-02 Variables

SH-02-1 Local Variables

Local variables in functions shall be declared local via whatever methods are available in that language (bash **`local`** or ksh **`typeset`**).

References

[c11] ISO/IEC, *Programming Languages - C*. ISO/IEC standard 9899:2011

[c89] ANSI, *Programming Language C*. ANSI standard X3.159-1989.

[c89-9899] ISO/IEC, *C Integrity*. ISO/IEC standard 9899/AMD1:1995.

[c99] ISO/IEC, *Programming Languages - C*. ISO/IEC standard 9899:1999.

[C++98] ISO/IEC, *Programming Languages - C++*. ISO/IEC standard 14882:1998.

[C++03] ISO/IEC, *Programming Languages - C++*. ISO/IEC standard 14882:2003.

[C++11] ISO/IEC, *Programming Languages - C++*. ISO/IEC standard 14882:2011.

[C++14] ISO/IEC, *Programming Languages - C++*. ISO/IEC standard 14882:2014.

[C_JPL] JPL Institutional Coding Standard for the C Programming Language, http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf, (JPL/CalTech, 2009)

[MISRA04] Motor Industry Software Reliability Association (MISRA), MISRA-C: 2004, Guidelines for the use of the C language in critical systems, October, 2004.

[f90] ISO/IEC, *Information technology -- Programming languages -- FORTRAN*. ISO/IEC standard 1539:1991.

[f95base] ISO/IEC, *Information technology -- Programming languages -- Fortran - Part 1: Base language*. ISO/IEC standard 1539-1:1997.

[f95str] ISO/IEC, *Information technology -- Programming languages -- Fortran -- Part 2: Varying length character strings*. ISO/IEC standard 1539-2:2000.

[f95cond] ISO/IEC, *Information technology - Programming languages - Fortran - Part 3: Conditional compilation*. ISO/IEC standard 1539-3:1998.

[f95mtclf] M. Metcalf and J. Reid. Fortran 90/95 Explained. 1999.

[photran] Photran - An Integrated Development Environment for Fortran. https://wiki.eclipse.org/PTP/photran/documentation

[f95crtm] Paul van Delst, CRTM: Fortran95. Coding Guidelines Joint Center for Satellite Data Assimilation (internal), JCSDA/EMC/SAIC, January, 2008.

[f95gsi] Paul van Delst, GSI Code Standard. Last update 2016 (internal), EMC/IMSG, https://svnemc.ncep.noaa.gov/trac/gsi/wiki/GSI%20code%20standards.

[f2003base] ISO/IEC, *Information technology - Information technology -- Programming languages -- Fortran -- Part 1: Base language*. ISO/IEC standard 1539-1:2004.

[f2003base] ISO *Enhanced module facilities in Fortran*. ISO technical report *TR-19767:2005*.

[fortranwiki] FortranWiki, *Fortran 2008 Status*. http://fortranwiki.org/fortran/show/Fortran+2008+status

[ieee-posix1] IEEE and The Open Group, The Open Group Base Specifications Issue 7 IEEE Std 1003.1™-2008, 2016 Edition http://pubs.opengroup.org/onlinepubs/9699919799/

[gnu-style] GNU Project, *GNU Coding Standards* https://www.gnu.org/prep/standards/standards.html

[ncepee2] NCEP, *Environmental Equivalence Version 2* In review.

[ncepimpl] NCEP Central Operations - WCOSS Implementation Standards
http://www.nco.ncep.noaa.gov/idsb/implementation_standards/
[perlstyle] Larry Wall, *Perl Style Guide*. Perl manual page "perlstyle," distributed with the perl distribution.
[pythongoogle] Google, *Python Style Guide* version 2.59.
https://google.github.io/styleguide/pyguide.html
[refactor] M. Fowler. Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

Additional Links:

[ESMF] ESMF Software Developer's Guide

http://www.esmf.ucar.edu/esmf_docs/dev_guide.pdf

[WRF] WRF Coding Conventions

http://www.mmm.ucar.edu/wrf/WG2/WRF_conventions.html