# ASTG Coding Standard

## Recommended Coding Styles for Software Development in Fortran, C++, Java, and Python

**Version 1.7**

ASTG

January 20, 2015

# Contents

# 1 Introduction

## 1.1 Purpose Of Conventions

This coding standards document is intended to provide guidance for the selection of names, formatting of structures, use of comments and other issues in the development of programs.

## 1.2 Why Use Conventions

Maintenance is by far the largest development cost in the software's lifetime. The original author of software rarely maintains it indefinitely, so the use of good coding conventions can provide benefits that payoff in the future. Conventions improve readability and maintenance by making code more uniform, standard and comprehensible by different readers.

While a good development environment (IDE), such as Eclipse, can improve the readability of code by automatic formatting, the programmer should never rely on such features. Source code should always be written in a way that maximizes its readability, independent of any IDE.

*"Programs must be written for people to read, and only incidentally for machines to execute."* —

Abelson & Sussman, **Structure and Interpretation of Computer Programs**

## 1.3 When NOT To Use These Conventions

1. **Customer's preference:** The coding style for a legacy code shall be followed when the code is owned by another organization that requires the use of their style. Inquire before assuming that we may or may not use our coding standard.

2. **Existing code:** Existing legacy code shall not be converted to fit this standard. Extensive rewriting of legacy code to match a new style takes time and can introduce new bugs. The use of automated tools is preferred if a style conversion is desired. Legacy code that is being modified shall also not be updated to fit this standard. Mixed, inconsistent styles within a file can be more confusing than a weak style alone. An exception is made if the change is easy, consistent throughout and improves the readability of the code. In some cases the alterations are significant enough to rewrite a large portion of the file. When that occurs, a judgment call should be made by the developer to convert the entire file to the new style.

3. **Extending a framework:** A framework that is extended from a third party shall use the naming conventions used in that framework. This maintains a basic consistency with that framework and clearly indicates that the code is part of that framework. However, other conventions from the coding standard that do not necessarily have to do with names can be followed in this case.

## 1.4 When To Use These Conventions

These coding conventions shall generally apply only to code that is newly developed.

1. **New files:** This coding standard shall only be followed when starting new files. This applies to both new projects and significant modifications to legacy code. Each file can be treated as a consistent unit of coding style.

2. **Simple changes:** Good programming practices (e.g. such as using **`implicit none`**) may be applied to files when the change is simple. Additionally, if the codebase is relatively small at this point, it could make sense to convert it to the new style.

# 2 Files

## 2.1 Naming Files

### 2.1.1 Use Class/Module Name For File Name

- File names shall generally use the same name as the class/module they implement.
- Files with multiple classes or modules per file should be avoided for most cases.
  *Python*: A file may include more than one class if these additional classes have no internal methods (used strictly for setting up data structures).
- Unit testing files, used in frameworks such as pFUnit, JUnit, or CppUnit, must include the "test" prefix required by the framework.
- *Fortran*: Fortran filenames shall include the "_mod" suffix from the module.

### 2.1.2 Extensions

Source file extensions shall be named according to the following list. Exceptions apply in cases where the compiler or operating system has issues with the convention or the original project uses another convention:

| | |
|---|---|
| C source file | .c |
| C header file | .h |
| C++ source file | .cpp |
| C++ header file | .h |

| | |
|---|---|
| Fortran header file | .h |
| Original Fortran-90/95 source file (requires pre-processing of #directives) | .F90 |
| Processed Fortran-90/95 source file (without pre-processing directives) | .f90 |

| | |
|---|---|
| Java source file | .java |

| | |
|---|---|
| Python source file.py | .py |

**Examples:**

- Fortran module SomeModule_mod: `SomeModule_mod.F90`
  Associated test module: `testSomeModule_mod.F90`
- C++ class SomeClass: `SomeClass.h` and `SomeClass.cpp`

## 2.2 File Organization

Files shall be organized so that information appears in a standard order. The standard code templates are recommended for starting new code.

The following files shall contain the elements in this order:

### 2.2.1 C/C++ Header Files

1. Standard ASTG comment header
2. Inclusion protection directive
3. Include files
4. Global and file scope identifiers (except classes)
5. Class interface
6. Inline functions

### 2.2.2 C/C++ Source Files

1. Standard ASTG comment header
2. Include files
3. Global and file scope identifiers (except classes)
4. Class implementation

### 2.2.3 Fortran Files

1. Standard ASTG comment header
2. **`program, module, procedure, function`**
3. **`use module`** statements
4. **`implicit none`** declaration
5. **`private`** (as default; **`public`** entities are declared explicitly)
6. Include files
7. Variable declarations (dummy arguments may appear before includes, then locals)
8. Source code
9. **`contains`** block

### 2.2.4 Java Files

1. Standard ASTG comment header
2. Optional **`package`** directive
3. Zero or more **`import`** directives
4. Class implementation

### 2.2.5 Python Files

1. **#!/usr/bin/python** or **#!/usr/bin/env python** (preferred)
2. Standard ASTG comment header (docstring format preferred)
3. Zero or more **import** directives
4. **def** or **class**
5. Docstring* comment for all public routines using """triple double quotes"""
6. Code implementation

*Docstrings should explain **how to use** the code, whereas normal comments describe why and how the code works.

## 2.3 C++ Preprocessing

### 2.3.1 Minimal Includes

The header shall only contain the include files necessary to successfully compile the header in a particular source file. All other include files belong in source files. This lowers compile time and may reduce the size of the library.

### 2.3.2 Brackets Verses Quotes

Include files that are used as part of the same file package shall use the **#include "..."** form. External include files shall use the **#include <...>** form.

```
// MyClass.cpp

#include "MyClass.h"
#include <vector>
#include <sys/types.h>
```

### 2.3.3 Inclusion Protection

Include files shall use inclusion protection in the following manner:

```
#include "MyClass.h"

#ifndef MYCLASS_H
#define MYCLASS_H

// code here

#endif // MYCLASS_H
```

#### 2.3.4 Avoid Macros

Constants and inline functions shall be preferred over the use of a macro.

```
#define DEFAULT_NAME "Noname"
#define SET_NAME(name)
    // ...

// This is a better solution:
static const char* DEFAULT_NAME = "Noname";
inline void setName(const char* name) {
     // ...
}
```

# 3 Formatting

Code should be formatted in such a way that a new person can pick up the code and quickly understand it. The use of whitespace can improve readability by visibly separating sections of code.

## 3.1 Blank Space

### 3.1.1 Blank Spaces

Blank spaces shall be used in the following circumstances:

1. After a keyword that is followed by a parenthesis (but not after a method name)

```
while (true) {      // space after while
    doSomething();  // but not after function name
}
```

2. After a comma or semicolon in a loop or arguments in a list

```
int main(int argc, char * argv[])
do i = 1, 100        ! space after loop comma
for (i = 0; i < n; i++)
```

3. No space around primary operators

```
hypot[i] = pythag(wind.u, wind.v);
model->windspeed = hypot[i];
```

4. Between binary and assignment operators (but not required around the '=' when defining a default parameter value)

```
! spaces around =, + and *
newArea = ((len + lenIncrement) * width)
newVolume = doSomething(arg1, arg2, arg3=defaultValue)
```

5. Around conditional operators, except "!" (not)

```
maxDistance = (a \texttt{>} b) ? a : b;
proceed = !(maxDistance == 0);
```

6. After the unary operator in a declaration and before it when variable is used

```
char* firstName = "Bob";      // space after * not before
Object& o = new Object();     // space after & not before
*fullName = *firstName + " " + *lastName;
```

Blank spaces shall never separate increment ++ and decrement – operators from their operands

```
for (index = 0; index < maxNum; index++) {
    ...
}
```

### 3.1.2 Indentation

The following rules shall apply to indenting code:

1. Tabs shall not be used to indent code as the definition of tabs can vary across environments and can make code nearly unreadable. Configure your editor to replace tabs with spaces.
2. Spaces shall be used for all indentation to ensure the proper alignment of code between editors.
3. Each level of indentation shall be three (3) spaces for Fortran code and four (4) spaces for Java, C++, and Python code. The difference is mainly due to historical convention and the general consensus among the developer community.

## 3.2 Lines

### 3.2.1 Line Length

Lines should be limited to eighty (80) characters. This will allow readable printouts, plus, it enables side-by-side coding windows without disruptive line wrapping.

Lines may extend beyond 80 characters in special circumstances; however, lines that would exceed 132 characters must always be put on multiple lines.

### 3.2.2 Line Continuation

For lines that must be continued, developers may use their own style. The indentation for the broken lines shall at least be aligned on the left side. The preferred place to break around a binary operator is *after* the operator, not before it.

***Fortran:*** Continuation characters (&) shall be used *both* at the end of the last line and the beginning of the next line. The continuation characters should generally be aligned, when feasible.

```
print*,'This long statement is going to be continued on the  &
       &following line'}
```

***Python:*** Continuation characters (\) may be used at the end of the line, though Python's implied continuation inside parentheses, brackets, and braces is preferred.

```
print ('This long statement is going to be continued on the '
       'following line')
```

#### 3.2.3 Single Blank Lines

Single blank lines should be used in the following circumstances:

1. Before comments
2. Between methods
3. Before logical sections of code inside a routine

#### 3.2.4 Double Blank Lines

Double blank lines should be used in the following circumstances:

1. Between classes, interfaces and modules
2. Between the major sections of a source file

### 3.3 Fortran Free-Format

Code shall be written in free-format syntax. This is the preferred way to code in modern languages since it is easier to format naturally and read.

## 4 Naming

*The use and arrangement of identifiers is similar to comments because it makes code easily recognizable and readable. Self-describing code needs fewer comments because the code explains itself.*

### 4.1 Expressiveness and Scope

Identifiers with larger scope shall have more expressive names since they are useable in a larger body of code. Using i, j, k for temporary variables in **for**/**do** loops is generally acceptable when the loop is not long.

```
subroutine recordDatabase(i, currentRec)
    integer, intent(in)  :: i          ! large scope, not acceptable!
    integer, intent(out) :: currentRec ! much better
```

### 4.2 Abbreviations

#### 4.2.1 Avoid Uncommon Abbreviations

Abbreviations, especially for class and types, shall be avoided since they are more difficult to recognize. Commonly used abbreviations are acceptable, such as: min, num, temp, etc.

### 4.2.2 Use Mixed Case Acronyms

- Acronyms should be avoided if at all possible.
- Use all upper case letters for the acronym. Put underscores between the acronym and other capital letters.
- If the identifier needs to start with a lower case letter, such as in a variable name, then use all lower case letters for the acronym. Do not use an underscore after the acronym.
  Examples:

```
ESMF_URL_String
getXML_String()
esmfFilename
```

## 4.3 Case and Underscores

- Identifiers (and keywords) should be named consistently across programs and among developers.
- Underscores should be used only when necessary, such as in the use of all capital letters in parameters (MAX_NAME_LENGTH) or when the term may become unclear.
- *Fortran*: Optional parameters with default values shall use an underscore at the end to differentiate the local variable used to assign it a value.

```
subroutine foo(someValue)
   integer, optional, intent(inout) :: someValue
   integer :: someValue_

   someValue_ = defaultValue
   if (present(someValue)) someValue_ = someValue
   ...
end subroutine foo
```

- *C++ or Java:* Conflicting variable names shall be avoided. Using a local variable or method parameter with the same name as a class's member variable can be confusing. If this is preferred on occasion, such as in the use of a constructor or mutator parameter, then use an underscore at the end of the parameter.

```
Class SomeClass {
    public:
        // Avoiding the same parameter:
        void setSomeNumber(int initSomeNumber) {
            someNumber = initSomeNumber;
        }
        // Using an underscore:
        void setSomeNumber(int someNumber_) { someNumber = someNumber_; )
    private:
        int someNumber;
};

```

## 4.4 Naming Rules For Identifiers

The following table describes how different program elements shall be named:

| Identifier | Naming Rule | Examples |
|---|---|---|
| Keywords | Use all lower case letters | `program, for, switch` |
| Modules | Mixed case letters that use capitals letters for each word. (**Fortran**: end with the suffix "_mod".) | `ShallowWater_mod`<br>`LightningPhysics_mod` |
| Namespaces | Mixed case, starting with capitals letters for each word. | `Math`<br>`GridToolKit` |
| Classes, Interfaces, and Exceptions | Mixed case, starting with capitals letters for each word. | `SystemPrinter`<br>`GridPoint`<br>`ESMF_Initialize`<br>`NullPointerException` |
| Test Modules and Classes | Test code should start each name with "test" to remain compatible with unit testing frameworks, such as CppUnit and pFUnit. | `testGridPoint`<br>`testShallowWater_mod`<br>`testAsyncIO_mod` |
| Variable Types | Mixed case, starting with capital letters for each word (don't end with "_type") | `PrintCode`<br>`GridBoundary` |
| Methods and Subroutines | Mixed case, starting with a lower case letter. First word is normally a verb. | `getJobCount()`<br>`hasNextRecord(numRec)` |
| Test Methods & Subroutines | Unit testing framework methods should use mixed case, prefixed with "test". | `testGetJobCount()`<br>`testHasNextRecord()` |
| Variables | Mixed case, starting with a lower case letter. | `jobCount`<br>`currentJobNum`<br>`gmiURL_String` |
| Constants | Constant values shall be all capitals, with underscores between words. | `MAX_JOBS`<br>`BUFFERS_AVAILABLE` |
| Enumerations | All capitals, with underscores between words. | `COLOR_RED`<br>`JOB_STATUS_NORMAL` |
| Directives | All capitals, with underscores between words. Pre-existing directives, such as OpenMP, will comply with their own required style. | `HAS_FORK` |
| **Python only**: Private routines and variables | Use a leading underscore (single or double) for internal variables, methods, and functions to protect from being imported. | `_someNumber`<br>`__internalFunction()` |

Table 1: Naming rules

# 5 Declarations

## 5.1 Routines

### 5.1.1 Verbs for Routines

Routines shall begin with a verb, preferably a strong action verb. Avoid the use of generic terms, like "process" by being more specific about what the process actually is doing.

```
1 subroutine parseMessage(inputMessage)
```

### 5.1.2 Accessors and Mutators

Accessor and mutator functions shall begin with **`get`**, **`set`**, and **`is`**. Optionally, **`has`** may be used for testing Boolean possession, as opposed to a Boolean condition.

### 5.1.3 Avoid Global Routines

Routines that are global can often be placed inside of a related class or module to avoid naming collisions.

*C++ and Python*: They might also be placed in a namespace. This avoids polluting the global namespace.

## 5.2 Variables and Arguments

### 5.2.1 Use Nouns

Variables and arguments shall be named with nouns since they represent a thing or quantity.

### 5.2.2 Proximity of Declaration

*C++ and Java*: Variables shall be declared just before the block of code where they are used, instead of the top of the method.

```
void updateRecord() {
   bool isDone = false;  // don't declare here, before isDone is needed!
   //...

   bool isDone = false; // better
   while (isDone == false) {
      //...
```

### 5.2.3 Initialize Variables

All variables must be explicitly initialized before use, avoiding problems with the assumed value of uninitialized variables.

*C++, Java, and Python*: Variables shall always be initialized where they are declared, or shortly thereafter.

### 5.2.4 Zero Pointers

Pointers that have not been allocated or have been deallocated in C++ shall be set to a null value. This reduces double delete and wild pointer bugs.

### 5.2.5 One Declaration Per Line

Multiple declarations per line shall be avoided, unless the variables are very tightly coupled. The use of multiple lines encourages comments and improves readability.

```
integer :: sendCount, recvCount      ! okay, tightly coupled
integer :: printJob
```

### 5.2.6 Avoid Literals

Constants shall be used instead of literal constants. Literals (a.k.a. Magic Numbers) in the code should be avoided. An exception is made for -1, 0, 1 and 2 if used as looping values or in basic math.

### 5.2.7 Use Booleans for Booleans

Boolean values shall be used when available in the language, rather than "0" or "1". Constants shall be defined as "true" and "false" when a Boolean type is unavailable in the language.

### 5.2.8 Descriptive Namelist Variables

*Fortran:* Namelist variables shall avoid using numeric literals (with the possible exception of 0 and 1). Strings are preferred since they provide a much better description of the options.

### 5.2.9 Avoid Global Variables

Global variables shall not be used. There are times when globally accessible variables are useful. However, global variables hinder code reuse, are difficult to modify access (such as adding error checking) and are not easily converted into an abstract type.

Use a function inside a module, related class, or namespace instead of a global variable.
Refer to the *Singleton* pattern if a global class is needed.

```
// Don't use globals like this!
static Printer* g_Printer = new Printer();

// Instead you might use an accessor that wraps the global.
inline Printer* getPrinter() {
    // NOTE: inline statics only ever create one instance
    static Printer* printer = new Printer();
    return printer;
}
```

### 5.2.10 Use Portable Numeric Types

*C++*: The use of portable type definitions shall be used in programs to enhance portability. Unfortunately, C-style numeric types like **`int`** can represent any size value (16 bit, 32 bit, 64 bit, etc.)

Use "**inttypes.h**" file types (e.g. **`int32_t`**) to maintain platform-independent implementations.

### 5.2.11 C++ Casting

#### Avoid Casting

The use of **`void*`** in the public interface functions shall be avoided. It is acceptable to use them in the implementation, but they should generally be avoided if possible. Instead, prefer the use of templates to obtain the benefits of type safety.

#### Use C++ Style Casts

The C++ style operators (**`static_cast`**, **`dynamic_cast`**, **`const_cast`**, **`reinterpret_cast`**) shall be preferred in situations requiring casting. These operators offer significantly more precision and safety in operations requiring casting.

### 5.2.12 Use Argument Modifiers

*C++*: The modifiability of each argument passed by reference (including pointers) shall be specified using **`const`** for read-only variables.

*Fortran:* The *intent* of each argument (i.e. `in`, `out`, `inout`) shall be specified before each argument declaration in a routine.

```
subroutine updateSurfaceAreaDensity(initNum, finalCond)
   implicit none
   integer, intent(in ) :: initNum
   real*8,  intent(out) :: finalCond

   ! ...

end subroutine updateSurfaceAreaDensity
```

*C++ and Java:* A similar system (recommended) for showing *intent* can be achieved by putting each argument on a separate line:

```
void updateSurfaceAreaDensity(int   initNum,    // in
                              float finalCond)  // out
{
     // ...

}
```

### 5.2.13 Use "Implicit None"

*Fortran:* **implicit none** shall be at the top of all program units to ensure that variables are explicitly declared, documented, and type checked. It is the default in module functions if declared at the top of the module.

### 5.2.14 Automate I/O "Unit Numbers"

*Fortran:* As of Fortran 2008, the language provides a **`newunit`** specifier to the **open** statement, which shall be used to obtain I/O unit numbers, instead of hard coding constants such as the numbers 5 and 6. This **`newunit`** intrinsic automatically assigns a unique negative unit number, preventing conflicts with any existing unit numbers.

```
open(newunit=myUnit, file='surface_data.txt', ...)
read(unit=myUnit, iostat=ioerr) sfcTemp
```

## 5.3 Classes, Modules and Types

### 5.3.1 Use Pure Block Layout

*Pure block* layout shall be used to layout a class or module. Pure block uses (1) one line to designate a block opening, (2) one or more indented lines for the internal block and (3) one line for closing the block.

```
subroutine getIntersectionBounds(this, boundingBox, intersectionBounds)
   type(MyCustom_type), intent(in) :: this
   integer, intent(in) :: boundingBox(2, NUM_DIMENSIONS)
```

```
    integer, intent(out) :: intersectionBounds(2, NUM_DIMENSIONS)
    !...

end subroutine getIntersectionBounds
```

The braces-stand-alone style is preferred for C++, Java, and Python, though placing the brace at the end of the block opening line is fine too.

```
class BankAccount            // putting the '{' up on this line is also acceptable
{
    public:
        float getBalance();

    //...
};
```

Class and module modifiers, like **public**, **private** and **implicit none**, shall also be indented, with the exception of **contains**.

```
module BankTransaction_mod
   public creditAccount
   implicit none}

contains
   subroutine creditAccount(account)
      !...
   end subroutine creditAccount}
end module BankTransaction_mod}
```

In some cases, an **if** block or **for** block may have a single statement that doesn't technically require a closing statement/brace. Nevertheless, a closing brace or statement shall *always* be used, for both clarity and maintainability (except in Python).

```
// Never do this:
for (row = 0; row < gridMax; row++)
    if (fieldValue < 0)
       fieldValue[row] = 0;
```

```
// Much better:
for (row = 0; row < gridMax; row++)
{
    if (fieldValue < 0)
    {
        fieldValue[row] = 0;
    }
}
```

### 5.3.2 Fortran Derived Types

#### Standardized Constructor and Destructor Names

Creating a variable of a derived type loosely corresponds to creating an object of a particular class in C++/Java. This is a desirable feature from object-oriented languages, and as of Fortran 2003, constructors with the same name as the derived type are supported, though the corresponding destructor syntax is not. The following

naming conventions shall be used for derived type constructors and destructors. Note that a package-name prefix in front of the module name is highy encourged, to avoid possible naming conflicts.

Type name: Foo

Module name: [package_name_]Foo_mod

File name: [package_name_]Foo_mod.F90

Constructor interface name: Foo

Constructor name: newFoo

Destructor name: destroyFoo

```
module Foo_mod
   ...
   type Foo
      integer :: windDirection   ! degrees, clockwise from North
      real    :: windSpeed       ! in kts, 6m above the surface
   end type Foo

   interface Foo
      module procedure newFoo
   end interface

contains

   function newFoo() result(this)
      type(Foo) :: this
      ! Initialize data values
      this%windDirection = 0
      this%windSpeed = 0

    end function newFoo
    ...
end module Foo_mod
```

If any memory had been allocated in the Foo constructor, the destructor destroyFoo would be used to deallocate that memory.

PLEASE NOTE: Fortran compilers pre-2003 do not support routine names that are the same as derived-type names. These legacy compilers must use constructor interface names with a "create" prefix, such as "createFoo", and a constructor with the name "newFoo".

### 5.3.3 Use Lowest Possible Scope

Elements within an abstract type (e.g., variables and routines) should kept be as private as possible, while still allowing the users of those items to perform their work. Public member variables of a class or module shall be avoided. Instead, **get** and **set** methods should be used to access and change the variable outside the class or module. This enables the benefits of encapsulation.

This also applies to Fortran derived types, which shall use the **private** modifier to restrict the public interface of a type.

```
module BaseTime_mod
   implicit none
   private
```

```
4
   public :: BaseTime
6
   type BaseTime
8     private                    ! good - explicitly declare public elements
      integer*8 :: wholeSeconds  ! only this module can read and modify wholeSeconds
10  contains
      procedure :: set           ! use these 3 procedures to modify and access the
12    procedure :: get           ! private variable
      procedure :: add
14  end type BaseTime

16 contains

18  subroutine set(this, wholeSeconds)
      integer*8, intent(in) :: wholeSeconds
20    this%wholeSeconds = wholeSeconds
    end subroutine set
22
    integer*8 function get(this) result(wholeSeconds)
24    class (BaseTime), intent(in) :: this
      wholeSeconds = this%wholeSeconds
26  end function get

28  subroutine add(this, increment)
      class (BaseTime), intent(out) :: this
30    integer*8, intent(in) :: increment
      this%wholeSeconds = this%wholeSeconds + increment
32  end subroutine add

34 end module BaseTime_mod
```

### 5.3.4 Classes

#### Group by Accessibility, then Functionality

Classes, modules and members shall be primarily grouped with the most public elements appearing earliest in the file and the most private elements appearing later in the file. Since the public interface is the most important it should be easiest to locate. Related elements should also be grouped according to functionality.

Classes or methods within a class or module shall be grouped according to functionality. This is consistent with keeping related and tightly coupled code grouped together.

#### Use a Default Constructor

A default constructor shall always be provided. Classes that do not support default construction should declare the constructor **protected** or **private**.

#### C++ Standard Copy and Assignment

Copy constructors and assignment operators shall use the argument "**other**" to identify the object being copied.

```
class A {
2     public:
          A(const A\& other);
4         A& operator=(const A\& other);
};
```

#### C++ Check for Self Copy

The assignment operator shall always check for self-copy. This can cause problems when the object deallocates memory in preparation for copy.

```
A& A::operator=(const A& other) {
   if (this != other) { // avoid self-copy problems!
      // ...
   }
   return this;
};
```

#### Operators

Overloaded operators shall be avoided in most circumstances. They can often add confusion.

#### Virtual Members

The destructor shall be declared **virtual** once any function becomes virtual.

The **virtual** modifier shall always be repeated in virtual functions for members of subclasses. This makes it clear to the reader that the function is actually virtual.

## 5.4 Use of External Units

### 5.4.1 Use Explicit Namespace or Use-Module Items

When introducing items from a namespace or module, prefer to explicitly identify the entity you want to use. This helps to easily identify the origin of the item. If the module is commonly used (such as the C++ "std" namespace), or there are a large number of items to use, then it may be preferable to include the whole unit.

```
use Foo_mod, only: startBar1, stopBar     ! good, identifies methods precisely
use Foo_mod                               ! bad, where do Bar methods come from?
```

### 5.4.2 When To Use Namespaces

*C++*: User-defined namespaces shall be used in larger programs to:

- Simplify the use of names in a program
- Prevent name clashes (e.g. enumerated types that use the same value)
- Group related program components (e.g. classes, global functions) into logical modules

### 5.4.3 Using "using namespace" Clause

*C++*: The **using namespace** clause shall not be used in a header file. This pollutes any source files that use the header with the entire namespace. Instead, use the "::" scope operator with the actual namespace.

# 6 Structure and Control

## 6.1 Statements

### 6.1.1 One Statement Per Line

Multiple statements per line shall be avoided. The use of one statement per line gives a better indicator of program complexity, makes it easier to follow top-to-bottom, trace execution and locate compiler errors.

### 6.1.2 One Operation Per Statement

Statements that try to perform several operations into a single statement, such as a side-effect, shall be avoided. These make it more difficult to modify code and more difficult to understand.

```
// side effects! -- errorCode and recNum are changed during a conditional
if ((errorCode = insertRecord( ++recNum )) == SUCCESS)
   //...

// solution without side-effects
++recNum;
errorCode = insertRecord(recNum);
if (errorCode == SUCCESS)
   //...
```

This does not mean that a chain of math operations must be one math operation per line. Math operations should be kept together if they are readable.

## 6.2 Blocks

### 6.2.1 Pure Block Layout

*Pure block* layout shall be used to layout a block of statements. Each sub-block shall be indented another level Refer to section 5.3.1 for a description of pure block layout.

```
do i = 1, 100
   statement1
   statement2
   if (statement3) then
      print*, statement4
   end if
end do
```

### 6.2.2 Named Blocks

Labels should be used for longer blocks of code to provide clarity, especially if there are multiple inner loops. *Fortran2008*: Labels also provide an elegant method to exit an outer block from the middle of an inner block.

```
Outer: block
   InnerLoop: do i = 1, 5
      ...
      if (x > X_MAX) exit Outer
      ...
   end do InnerLoop
   call someRoutine
```

```
end block Outer
```

### 6.2.3 Dummy Body Blocks

If a loop statement has a dummy body, opening and closing brackets should be added on separate lines. It is good practice to add a comment stating that the dummy body is deliberate.

```
for (int i = 0; i < numRecs; i++)
{
    // dummy body
}
```

### 6.2.4 Directives

Directives shall be indented with the code.

```
void displayDebugMessage(char* msg)
   writeToLog(msg);

   // also write to screen if debug mode
   #ifdef _DEBUG
   cout << msg << endl;
   #endif
}
```

### 6.2.5 Case Statements (except in Python)

#### Indent Under Cases

**switch** (**select** in Fortran) statements shall be formatted with each **case** statement indented one level and code for each case indented to a second level. This keeps code easily readable for each case.

```
switch (userAction) {
   case MOVE_MOUSE:
      updateMousePosition();
      showMouseCursor();
      break;
   case EXIT:
      closeAllWindows();
      shutdownSystem();
      break;
}
```

#### Avoid Dropping Through a Case (N/A in Fortran)

Each case shall always have a break; **case** statements without a **break** statement fall through to the next case. This may seem useful in some circumstances, but the code is more difficult to understand because two different control constructs are overlapped. Code modification is also more difficult without breaking each of the existing cases.

#### Add a Default Case

In most situations, add a default case to ensure all error conditions are handled properly. There may be some instances where a default is unnecessary, but if there is a possibility the current *or future* code could create an error condition, a default case should be added.

**Keep Cases Simple**

Code under each case should not be complicated or lengthy. If it is, then consider moving code under the case to a routine and invoking it from the case statement.

## 6.3 Concurrency

### 6.3.1 OpenMP and MPI

- OpenMP code shall use **Default(none)** and explicitly declare modifiers.
- MPI code shall avoid the use of **MPI_COMM_World** in situations that allow grouped processes. This will simplify design and encourage inter-process safety.
- MPI code shall prefer **use MPI** over **include "mpif.h"** to aid in the automatic detection of wrong usage of the interface to the MPI library.

## 6.4 Exception Handling in Java and C++

### 6.4.1 Use Judiciously

Exceptions shall be applied only in cases where error conditions occur that would not happen normally. For example: reading the end-of-file from a file is not an exception since it is occurs normally. Exception handling in can be a complicated undertaking, especially in C++, however it offers a more elegant solution in many cases. Exceptions should generally be preferred over error code returns.

### 6.4.2 Catch by Reference in C++

Are exceptions caught by reference to avoid the question of deletion responsibility and prevent duplicate copying and slicing?

### 6.4.3 Constructor and Destructor Issues

Exceptions shall not be raised inside the constructor of an exception as an infinite loop situation could occur. Likewise, destructors shall not let exceptions leave the destructor.

### 6.4.4 Re-throw the Same Exception in C++

Exceptions shall be re-thrown using plain **throw** as opposed to **throw anException** to avoid recopying the object.

```
try {
    //...
} catch (SomeException& ex) {
    throw;     // good
    throw ex;  // causes a copy to be thrown
}
```

### 6.4.5 Exception Handling in Java

Since Java will always throw an exception when encountering an error, each error should be caught to provide useful error messages. Java also includes an optional **finally** keyword, which will *always* be executed, and can assist in necessary cleanup.

```
try {
    someResource.UseResource();
} catch (MyException e) {
    // handle my exception
} catch (Throwable e) {
    // handle all other exceptions
} finally {
    someResource.dispose(); // special cleanup
}
```

## 6.5 Prohibited Fortran Features

Certain Fortran features have been deprecated by the Fortran-90 standard and are deemed to be poor programming practices. The following list of features shall be avoided:

| Feature | Replacement |
|---|---|
| **common** blocks | Modules |
| **Equivalance** | Derived data types<br>Use of **pointers** is okay, but generally avoid.<br>This feature may be useful in situations. |
| Assigned and computed **go to** | **case** construct |
| Arithmetic **if** statements | block **if** construct |
| Numeric statement labels (exception: block-style labels for loop control are permitted) | |
| **Pause** | |
| **entry** statements | |
| All other deprecated features | |

# 7 Comments

*Comments aid the reader of the code. Comments can be used to (1) summarize a section of code, (2) describe the intent of the code or (3) indicate something special about the code. There are no hard-and-fast rules for how many comments must be used. Self-documenting code that uses clear variable names and good formatting would require fewer comments.*

## 7.1 Embedded Comments

Programs shall use embedded comments as appropriate for extraction by a tool. This makes it much easier to keep the documentation consistent with the source code. The following tools shall be used:

- Fortran: Doxygen
- C++: Doxygen

- Java: Doxygen or Javadoc
- Python: Doxygen (can parse docstring format)

Doxygen has become the tool of choice, due to its versatility and widespread popularity. Doxygen will extract comments and tags from the source code and create hyper-linked reference documentation in multiple formats. In C++ or Java, a simple class prologue might look like this:

```
/**
 * @brief Example class to demonstrate basic Doxygen usage
 * @author I. M. Coder
 *
 * This is a simple class to demonstrate how Doxygen is used.
 * It shows how the special double asterisk comment begins a comment
 * block, and how both short and long descriptions are defined.
 */
```

Complete documentation is available at www.doxygen.org. For an examples of usage here at ASTG, refer to the code templates in Modeling Guru's Software Development community, or see Using Doxygen with Fortran Source Code.

## 7.2 Comment Major Code

The public interface should be well documented for those trying to use the code, while the implementation should be well documented for those trying to maintain the code. Examples of how to properly use code can be helpful if it is not obvious. Obvious code, such as getter and setter methods, does not necessarily need to be commented.

Comments and rules shall apply for the following table of major code items:

| Code | Placement Rule | Description Rule |
|---|---|---|
| File, Module | Top | Standard comment heading describing the purpose of the code |
| Class | Before the class definition | Description of the purpose of the class<br>Indicate the level of thread safety if appropriate |
| Routine, method | Before the routine implementation and throughout the routine | Description of the routine<br>Identify special pre-conditions and post-conditions, such as the deallocation of memory returned from the routine<br>Implementation and algorithm details go throughout the routine |
| Variable, member | Before or on the same line | Brief description of the variable, including units |

## 7.3 Use Descriptive Comments

Comments should add something useful to the code. They should normally describe blocks of code rather than an individual line.

- Comments shall not repeat code.

```
! update empRec                                  <-- poor comment
! get current employee information               <-- good comment
```

- Comments shall prefer to describe "why" or intent over "what" or paraphrasing.

```
! divide vector by length                        <-- poor comment
! normalize vector for use in transformations    <-- good comment
```

## 7.4 Avoid In-line Comments

- In-line comments shall generally be avoided, since they are harder to see, maintain and must be kept short.
- **Exceptions:** In-line comments are appropriate for short variable descriptions and for marking the ends of code blocks. The comments shall be aligned for each group of end-line comments.

```
                          ! comments are aligned:
integer :: gridWidth      ! number of parcels across
real*8  :: parcelSize     ! meters of each square
```

## 7.5 Proximity to Code

Comments shall be placed as close as possible to the code they describe. Comments that are closer are more useful to the reader and more likely to be maintained.

## 7.6 Indentation

Comments shall be written at the same level of indentation as the code they describe.

## 7.7 Special Comment Tags

The special comments listed below shall be used to tag code in certain conditions.

- **TODO** - code is incomplete and will be completed later. This code should indicate a warning when executed

```
do while (reportCount \texttt{>} 0)
   !TODO: process next report here
   print*, "TODO: process reports"
   reportCount = reportCount - 1
end do
```

- **FIXME** - code has a logical error and requires correction. This code should abort or throw an exception when executed.
- **NOTE** - a special circumstance applies and should be noted by anyone modifying or reviewing this section of code. This may indicate "surprise" code that does not work as expected or a deviation from the standard.

## 7.8 Python Docstring Comments

Docstrings, which are comments using """triple double quotes""", should be written for all public modules, functions, classes, and methods in Python. Docstring comments should appear immediately after the "def" line, and should describe code usage, including descriptions of the arguments and outputs. Implementation details should be put into standard block comments.

There are two forms of docstrings: one-liners and multi-line. The one-line version can be used for simple routines, and uses the same indentation as the rest of the function:

```
def factorial(n):
    """Returns the factorial of n"""
    while factor \texttt{<}= n:
        result *= factor
        factor += 1
    return result
```

The multi-line version begins with one line summary, followed by a blank line, and then a more elaborate description of the interface:

```
def readSurfaceTemp(fileName, minLat, maxLat, minLon, maxLon, tempArray):
    """Read in a portion of the surface temperature field from file

    Args:
    fileName: filename of temperature field, including full path
    minLat, maxLat: minimum/maximum latitude of field in degrees
    minLon, maxLon: minimum/maximum longitude of field in degrees

    Returns:
    tempArray: a 2-D array of temperatures within the given range of
               latitude and longitude

    Raises:
    IOError: An error occured accessing the given temperature file
    """
    ...
```

For multi-line docstrings, the final set of quotes must be on its own line.

A docstring can be extracted automatically through the __doc__() member of the object, and is available at runtime as an attribute of the object. Most IDEs will display them as context-sensitive "usage" tooltips, and Doxygen can extract them to auto-build documentation.

# 8 Practices

## 8.1 Non-Standard Conventions

*C++*: Non-standard language constructs and implementation dependencies beyond those specified by the ISO/ANSI C++ draft standard shall be approved by the team lead. This will reduce problems when code must be ported to another platform.

*Fortran*: The same principle applies to machine-specific features of Fortran, which should be avoided if possible. If this is not possible, the code shall be commented clearly to indicate the use of the machine-specific feature.

## 8.2 Block Copy

Copying blocks of code elsewhere in a program shall be avoided. Blocks of repeating code are an indicator of the need for a routine. Move the code into a routine and call the routine from each original location.

## 8.3 Portability

In general, practices that increase portability of code are highly recommended.

### 8.3.1 Intrinsic module iso_Fortran_env

*Fortran2008*: Use this module to obtain portable values for stdin, stdout, stderr, and "kind" values for various int and real variables.

```
use, intrinsic :: iso_fortran_env, only : &
                &  input_unit=>stdin, output_unit=>stdout, &
                &  error_unit=>stderr, &
                &  int8, int16, int32, int64 &
                &  real32, real64, real128
```

### 8.3.2 Execute Command-line Commands within code

*Fortran2008*: Use intrinsic routine execute_command_line to avoid non-portable solutions.

```
call execute_command_line(my_command                    &
                        & [,wait,exitstat,cmdstat,cmdmsg])
```

where:
`wait` – logical - if false, command will be executed asynchronously
`exitstat` – integer - exit status for synchronous commands
`cmdstat` – integer - indicates status of processor support for execution
`cmdmsg` – character - processor error message, if any

# 9 Appendix

## 9.1 References and Resources

### 9.1.1 Fortran Coding Conventions

*ESMF Software Developer's Guide*
*www.earthsystemmodeling.org/documents/dev_guide.pdf*

*WRF Coding Conventions*
http://www.mmm.ucar.edu/wrf/WG2/WRF_conventions.html

EnergyPlus Programming Standard
http://www.eere.energy.gov/buildings/energyplus/pdfs/programmingstandard.pdf

### 9.1.2 C/C++

*The Elements of C++ Style*
*Taligent's Guide to Designing Programs*
http://root.cern.ch/TaligentDocs/TaligentOnline/DocumentRoot/1.0/Docs/books/WM/WM_1.html

### 9.1.3 Java

*Sun's Code Conventions for the Java Programming Language*
http://www.oracle.com/technetwork/java/codeconvtoc-136057.html

### 9.1.4 Python

*Style Guide for Python Code*
http://www.python.org/dev/peps/pep-0008/

*Python Style Guide*
http://code.google.com/p/soc/wiki/PythonStyleGuide

*Note: ASTG standards differ somewhat from the above guides.

### 9.1.5 Documenting Code

Doxygen (C++, Java, Fortran, & Python)
http://doxygen.org

Javadoc (Java)
http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html