ANAG Code Standards

Applied Numerical Algorithms Group
NERSC Division
Lawrence Berkeley National Laboratory
Berkeley, CA

July 2, 2002

Abstract

Conventions for C++ and Fortran code, makefiles and documentation for ANAG, including unit and file organization, typographical rules, syntax preferences, class design, and code review.

Contents

Chapter 1

Introduction

This document defines C++ coding standards for code development for ANAG. The main purpose of ANAG code standards is to make it easier for programmers to understand each other's code. A secondary purpose is to help programmers avoid common pitfalls. The standards accomplish this by establishing conventional syntax usage and conventional ways of doing things.

ANAG coding standards are determined by programmer consensus and enforced by programmer buy-in, commitment to teamwork, and code reviews. We all agree to follow the standards whenever possible, to discuss exceptions to standards with other team members before implementation, and to clearly document the exceptions in the source code.

Chapter 2

General conventions

2.1 Source Code File Organization

Guidelines for naming, splitting, ordering, and organizing source code files.

2.1.1 Basic file organization

1. **File types.** Class and function declarations appear in header files, which have a `.H` extension. C++ Class and function definitions appear in source files, which have a `.cpp` extension, except for inlines. Inlined functions appear at the end of the header files in which they are declared. Chombo Fortran files have a `.ChF` extension.

2. **File names.** The filename is the name of most important class in the file. If the file contains only functions, then a descriptive name should be chosen. Filenames are capitalized `LikeThis.cc`, so that there is exact correspondence between the filename and the class it contains. In order to avoid name conflicts with other packages, don't use common names, like `Vector.h`.

3. **Source-header correspondence.** There is an exact correspondence between source and header files. E.g. if there is a `Function.H` there is a `Function.cpp`, and vice versa.

4. **Dividing code into files**. Files are divided along class boundaries. That is, if class `AmrLevel` is declared in file `AmrLevel.H`, all its member functions are defined in `AmrLevel.cpp`. Stand-alone functions follow the same rule. Several related classes may be grouped together in the same file pair. If either the header or the source file is too long (more than several hundred lines for headers, a couple thousand for sources), the file should be redivided, again along class boundaries.

5. **File headers**. Not to be confused with header files! The top lines of each file are comments stating

(a) Chombo logo (ASCII version),

(b) LBNL copyright.

E.g.,

```
/*   _______              __
    / ___/ /  ___  __ _  / /  ___
   / /__/ _ \/ _ \/  ' \/ _ \/ _ \
   \___/_//_/\___/_/_/_/_.__/\___/
*/
//
// This software is copyright (C) by the Lawrence Berkeley
// National Laboratory.  Permission is granted to reproduce
// this software for non-commercial purposes provided that
// this notice is left intact.
//
// It is acknowledged that the U.S. Government has rights to
// this software under Contract DE-AC03-765F00098 between
// the U.S.  Department of Energy and the University of
// California.
//
// This software is provided as a professional and academic
// contribution for joint exchange. Thus it is experimental,
// is provided ‘‘as is’’, with no warranties of any kind
// whatsoever, no support, no promise of updates, or printed
// documentation. By using this software, you acknowledge
// that the Lawrence Berkeley National Laboratory and
// Regents of the University of California shall have no
// liability with respect to the infringement of other
// copyrights by any part of this software.
//
```

Information about author(s), creation date, modification date(s), etc. can be obtained by looking at the CVS log, e.g., `cvs log ...`.

2.1.2 Header files

Header files contain the following items:

1. **Logo/Copyright comments.** See above.

2. **The multiple-include preventer.** The body of the header file is enclosed in a conditional preprocessor directive of the form

   ```
   #ifndef EBAMR_H
   #define EBAMR_H
   ```

for the file `EBAmr.H`. The preprocessor variable name is the filename in all capitals and period replace with an underscore.

3. **Include statements.** System headers are included like this:

```
#include <iostream>,
```

ANAG headers are included like this:

```
#include "DiscPDO.h".
```

Header files should include as few other headers as possible. Forward-declaration of other classes and inclusion of the header from the `.cpp` file are the alternatives.

2.1.3 Source Files

The source file contains definitions of all the functions declared in the header (except for the inlines, which are already in the header), and in the same order as declared in the header. The basic structure of a source file called `EBAmr.cpp` is

```
/*   _______              __
    / ___/ /  ___  __ _  / /  ___
   / /__/ _ \/ _ \/  ' \/ _ \/ _ \
   \___/_//_/\___/_/_/_/_.__/\___/
*/
//
// This software is copyright (C) by the Lawrence Berkeley
// National Laboratory.  Permission is granted to reproduce
// this software for non-commercial purposes provided that
// this notice is left intact.
//
// It is acknowledged that the U.S. Government has rights to
// this software under Contract DE-AC03-765F00098 between
// the U.S.  Department of Energy and the University of
// California.
//
// This software is provided as a professional and academic
// contribution for joint exchange. Thus it is experimental,
// is provided ‘‘as is’’, with no warranties of any kind
// whatsoever, no support, no promise of updates, or printed
// documentation. By using this software, you acknowledge
// that the Lawrence Berkeley National Laboratory and
// Regents of the University of California shall have no
```

```
// liability with respect to the infringement of other
// copyrights by any part of this software.
//

#include ``EBAmr.h''
#include <iostream>

// ------------- EBAmr -----------------

<EBAmr member functions>


// ------------- AnotherClass -----------------

<AnotherClass member functions>
```

2.2 Code Review Process

This section describes the "Code Review" process used by ANAG. Note, this process is not currently being used but will be the process used should code reviews once again occur.

2.2.1 Goals

The goals of the process are:

1. To assure that more than one pair of eyes has looked at each line of ANAG code. This means catching bugs, suggesting algorithm or data structure improvements, adherence to coding standards, etc.

2. To communicate the understanding of the code to a broader group of programmers, and hence to have better integration of components of ANAG software products.

3. Both members of a code review team "sign off" on the code, assume responsibility for it. If bugs need to be fixed, either will be able to fix them.

4. To aid in the creation of documentation for the code. One result of the code review process will be a description of each piece of code reviewed, which will be incorporated into the code source files.

5. To develop documentation (a permanent paper file) on the code design process on a per project/module/file basis.

2.2.2 The process

1. Reviewee prints code to be reviewed, and passed it to Reviewer.
2. Reviewer examines the code over the course of 24 hours, if possible,

   developing:

   (a) an understanding of the code

   (b) a written list of questions about the code

   (c) a written description of the code, and commentary, including suggestions for improvements in code as well as documentation

3. Reviewer and Reviewee sit together with printed source, discuss the questions and commentary, and come to a common understanding of the code, what changes need to be made in code and documentation
4. Reviewer make changes to code based on agreed upon understanding
5. Reviewee documents code based on agreed upon understanding
6. Reviewer incorporates documentation into updated code
7. Notes (formal or informal) from entire process are placed in code development folder.

Chapter 3

C++ Code

3.1 General Conventions

Typographical and design standards for classes, functions, and variables, and anything else that appears inside source code files. References in this section are primarily to *Effective C++* by Scott Meyers. This section under construction. Will be mostly references to or xeroxes of existing books. For now, I'll list the main points.

1. **Classes should be solid.** Note, this is a preference and is not adhered to everywhere in the Chombo code. Constructors should put objects into well-defined, working states. Non-const member functions should leave objects in well-defined, working states. In short, no object should ever be allowed to enter an ill-defined or non-working state. "Well-defined" means that all data members are initialized and have mutually consistent values (e.g. if the member `int length_` is meant to store the length of a member array at `Real* data`, the value of `length_` must equal the size of the allocated array at `data_`). "Working" means that all member functions execute without errors, unless the function arguments are outside the range of acceptable values.

2. **Classes should be minimal.** Classes should contain data members required to perform their functions, and *no others*. Extraneous data members complicate maintenance, and obfuscates the intended purpose of the code. Code becomes less self-documenting when there are non-critical data members hanging around. Also, the public interface should contain as few direct access to internal data members as possible. A public member function that returns a reference to the private data breaks encapsulation (the only *proven* benefit of OOP). The extraneous data members also make it hard to follow data through the system (which complicates future code revisions and testing).

3. **Do not rely on users to call a class member functions in a particular order** as a corollary of the previous rule. If this rule cannot be satisfied, then the condition

is documented. In the exception cases, member functions that must be called in a particular order shall be private functions with a public wrapper interface around them that calls them in the correct order. If this is not possible flags shall be inserted to make sure that the user did indeed call the functions in the correct order

4. **Classes manage their own memory, internally**. Users should not have to worry about it. In the cases where memory management is transferred, this must be boldly documented. There are only a couple of design patterns that require memory hand-off, and many of these can be robustified with more advanced programming techniques.

5. **Classes should be designed following established idioms.** Refer to a book before reinventing the wheel, or the ref-counting handle class. Designing around an existing idiom in a book saves time and provides ready-made documentation.

3.2 Lists of idioms

All references here refer to C++ Strategies and Tactics by Robert Murray.

1. **Ref-counting handle, no side-effects.** [Mur93] section 3.2. Allows objects to share data of variable size, but creates new copies of shared data when one object tries to change the data.

2. **Ref-counting handle, with side-effects.** Same as above with modified non-const member functions.

3. **Chesire-cat handle.** [Mur93] sections 3.3 and 3.4.

4. **Handle to a base class.** "Multiple implementations," [Mur93] sections 3.5.

5. **Smart pointers.** [Mur93] section 7.4.

6. **Templated container classes.** [Mur93] section 8.1.

7. **Iterators.** [Mur93] section 8.4.

3.2.1 Typographical standards

Indentation

1. **Do not use the tab character.** This does not mean you should never touch your tab button, only that before doing so you make sure it inserts spaces, rather than a tab character, into your file. The next section shows how to do that (if you use emacs).

2. **Use the emacs C++ editing mode.** This is to ensure that we're not continually overwriting each other's auto-indentation. A c++-mode hook to set the indentation-level can be placed in the programmer's .emacs file. The standard C++ is usually defined in `/usr/lib/emacs/lisp/c-mode.el` on the Linux machines. Where it usually is otherwise will vary. A sample `.emacs` file that autoloads this mode and changes the indent-level and tab-width is available is shown here:

```
(autoload 'c++-mode  "cc-mode" "C++ Editing Mode" t)
(autoload 'c-mode    "cc-mode" "C Editing Mode" t)

(require 'cl)
(setq auto-mode-alist
      (pairlis '("\\.cpp$" "\\.H$" "\\.CF$" "^Make." "\\.latex$")
       '(c++-mode c++-mode fortran-mode makefile-mode latex-mode)
              auto-mode-alist))

(add-hook 'c-mode-hook
          (function (lambda ()
                      (font-lock-mode 1)
                      (local-set-key "\M-\C-h" 'backward-kill-word)
                      (setq c-basic-offset 2
                            c-comment-only-line 0
                            continued-statement-offset 2
                            c-continued-brace-offset 0
                            c-brace-offset 0
                            c-brace-imaginary-offset 0
                            c-argdecl-indent 2
                            c-label-offset -2
                            c++-member-init-indent 2
                            c++-continued-member-init-offset 0
                            c++-empty-arglist-indent 2
                            c++-friend-offset 0
                            indent-tabs-mode nil
                            ))))

(setq c++-mode-hook c-mode-hook)
```

3. **Bracket placement.** Curly braces shall have their own line. Period. They may only share said line with a comment to say for what the curly brace is being used. For example,

```
 for(int ibox = 0; ibox < numboxes; ibox++)
```

```
{
  for(int idir = 0; idir < SpaceDim; idir++)
  {
    SideIterator sit;
    for(sit.begin(); sit.ok(); sit.next())
    {
      IVSIterator ivsit(ivs);
      for(ivsit.begin(); ivsit.ok(); ivsit.next())
      {
        blah(...);
      } // end loop over intvects in ivs
    } // end loop over sides in box
  } // end loop over directions
} // end loop over boxes
```

Names

1. **Class names are** `LikeThis`. That is, a class name begins with a capital, and words are demarcated by capitals rather than underscores.

2. **Object (variable) names are** `likeThis`, except for some traditionally capitalized mathematical objects. E.g. `Matrix A`.

3. **Function names are** `likeThis()`. This applies to class member functions and stand-alone functions.

4. **Err on the side of verbosity for all names.** The larger the scope, the greater the verbosity. Abbreviation is o.k. for large words (e.g. `Proc` for "Procedure"), and universally recognized acronyms are o.k. (e.g. `PDE` for "Partial Differential Equation"). Abbreviated names are spelled out completely in a comment preceding the declaration.

5. **Function arguments are named**. For example, this is o.k.: `int pow(int base, int power);`, but this is not: `int pow(int, int);`.

6. **Argument names are identical in definitions and declarations**.

7. **All modified arguments come before all unmodified arguments**. Default values are discouraged.

8. **Variable names follow the convention**:

   - **Member variables** begin with an `m_` as in `m_memVar`.
   - **Argument variables** begin with an `a_` as in `a_argVar`.

- **Static variables** begin with an `s_` as in `s_statVar`.
- **Global variables** begin with an `g_` as in `g_globVar`. Note, the use of global variables is heartily discouraged!

Miscellaneous

1. **Access levels in class derivation declarations are explicitly stated.** E.g. `class X : private Y` is o.k., `class X : Y` is not.
2. **Inline function definitions go outside class declarations**. For example,

```
class X {
public:
  int getDim() const;
private:
  int m_dim;
};

int X::getdim() const {return m_dim;}
```

3. **Ampersands and asterisks are attached to the base-type in declarations** rather than preceding the variable name. E.g. `X* x;` is o.k., `X *x;` is not.

3.2.2 General design standards

Miscellaneous standards

1. **Every time a rule is broken it must be clearly documented.**
2. **No global variables.**
3. **No memory leaks.** Not even small ones, since they obscure the large ones.
4. **No compiler-dependent assumptions.** E.g. on the size or layout of fundamental types.
5. **Declare variables in the smallest possible scope.**
6. **Every variable must be given a value before it is used.**
7. **Avoid side-effects**. I.e. for `X x; Y y(x); y.foo(); foo()` shouldn't change `x`. If it does it should be annotated to death in the declaration for `Y::foo()` and `Y::Y(X& x)`.
8. **Optimize code only if you know it's a performance problem.**

Preferences

1. **Use `const` and `inline` instead of `#define`.** ([Mey92] rule 1).
2. **Use symbolic variables rather than numerical values ("magic numbers") in code.** ([Mey92] rule 10)
3. **Use `bool` for boolean variables, not `int`.**
4. **Prefer streams to stdio,** for consistency and extensibility. ([Mey92] rule 2).
5. **Use `new` and `delete` instead of `malloc` and `free`.** An exception is when you really need to use `realloc` on a primitive type, since `new`/`delete` can't provide this functionality. ([Mey92] rule 3).
6. **Use `delete[]` (note the square brackets) when deallocating arrays.** ([Mey92] rule 5).

Pointers, references, and objects

1. **Specify all return types.** I.e. declare `int f(int n);` rather than `f(int n);`.
2. **Pass class objects by const-reference rather than by value.** ([Mey92] rule 22).
3. **The Linton convention:** Functions should not store pointers to *reference* arguments in any location that will persist after the function returns. Functions that need to do this should use *pointer* arguments instead. (C++ S&T p 214).
4. **Any reference function-argument that can be declared `const` is declared `const`.** Non-constness implies that the function changes the argument. E.g. `int f(const X& x)` does not change `x`, but `f(X& x)` changes `x`. ([Mey92] rule 21)
5. **Any pointer function-argument that can be declared `const` is declared `const`.** E.g. `int f(const char* s) {cout << s;}`. The `const` assures the user that the recipient won't alter attempt to delete the object.
6. **Prefer objects to pointers.** That is, if you can construct an object on the stack, like this: `X x(4,3);` rather than in the heap, like this `X* x = new X(4,3);` construct it on the stack. This leaves memory management to the compiler and results in simpler syntax. Often declaration with a default constructor, `X x;`, followed later by assignment, `x = X(4,3);` is a good substitute for declaring a pointer and then assigning it to a `new`'d object.

Errors

1. **Check the return value of `new` to see if it failed.** ([Mey92] rule 7).

3.2.3 Class design: the basics

This section includes some low-level ANAG class requirements and style tips. For more in-depth discussion of class design, see Section 3.1.

Required member functions.

Every class has an explicit version of

1. **A copy constructor,** `X::X(const X& x);`
2. **A destructor,** `X:: X();`
3. **An assignment operator,** `const X& X::operator=(const X& x);`

The default, compiler-generated, versions of these functions should not be used. If these methods are not to be used for a given class then private versions should be defined which generate an error message if invoked.

Almost every class has a

1. **A default constructor,** `X::X();`

There is no compiler-generated default constructor. Define one if you can so that others can make arrays of your objects, default-construct objects, and then assign into them, e.g.

```
X x;
if (test) {
  // do something
  x = X(results of something);
}
else {
  // do something else
  x = X(results of something else);
}
```

(though initialization is generally preferable to assignment).

Constructors

1. **Solid constructors** are preferred. Every constructor should result in a well-defined object, or it must fail, error and exit (or throw an exception). "Well-defined" means an object whose member functions work and whose member variables are internally consistent. For example, the null constructor for an int-array-like class result in a zero-length array, by setting its dimension member to zero and its pointer member to either 0 or the return-value of `new int[0];`. If the pointer is set to 0 then dereferences of the pointer (which would cause seg-faults!) in other member functions should be shielded by checks for pointer validity, or an equivalent check. The main point is that no object should be allowed to become a potential core dump.

2. **If a class has dynamically allocated memory, the copy constructor is always defined explicitly** and implemented to avoid leaks and unintentionally shared memory. ([Mey92] rule 11).

3. **The signature of the copy constructor is** `X(const X& x);` unless side-effects on copied objects are explicitly desired, in which case the reference is be non-const and the potential for side-effects is noted in comments at the declaration.

4. **The copy constructor results in an object that behaves identically to the copied object.** Usually this means all data members are copied.

Destructors

1. **Destructors are designed so as not to leak memory.** Usually that means they deallocate any memory allocated within the constructors, though this is not always true, for example some classes share memory and count references. ([Mey92] rule 6).

2. **Base classes have virtual destructors.** ([Mey92] rule 14).

Assignment operator

1. **The assignment operator results in a left-hand-side object that behaves identically to the right-hand-side object.** Usually this means all data members are assigned. ([Mey92] rule 16)

2. **The signature of the assignment operator is** `const X& operator=(const X& x);` unless side-effects are explicitly desired, in which case it is `const X& operator=(X& x);` and the potential for side-effects is noted in comments at the declaration. The returned `const X&` object is `*this`. ([Mey92] rule 15).

3. **The assignment operator checks for assignment to self,** with something like this

```
const X& X::operator=(const X& x) {
  if (this != &other)
    <do some work>
  return *this;
}
```

([Mey92] rule 17).

Memory management

1. **A class is in charge of managing its own memory.** A user should be able to create instances of the object, use them, modify them (especially by assignment), and let them go out of scope, without thinking about the class's internal memory management. For a good example, see C++ S&T section 9.3.1, page 215

2. **Prefer object members to pointer members.** This simplifies memory management and allows use of the compiler-generated copy constructor, destructor, and assignment operator. Of course, object members are not always appropriate (for example, when `A` needs access to a `B` whose lifetime is not constrained by `A`'s lifetime, `A` might be better with a const-pointer or reference member to a `B`.

Member data

1. **No public data members** is preferred. There should never be a public pointer member. ([Mey92] rule 20).

2. **Pointer members normally point to unshared memory allocated at construction and deallocated at destruction.** In cases where this is not so, a comment next to the pointer declaration should indicate non-ownership and the style of usage.

Member functions

1. **Any member function that can be declared `const` is declared `const`.** Non-constness implies that the function changes the object. If the behavior of an object is dependent on data outside the object, this data is not modified by const member functions ("conceptual constness", see ([Mey92] rule 21)).

2. **A pointer returned from a member function is assumed to be allocated on the heap, and the recipient is responsible for deleting it.** Exceptions are commented. It's preferable to avoid the issue by returning objects when possible.

Operators

1. **Unary operators are member functions of their classes.**
2. **Assignment-like binary operators (+=, *=, etc.) are member functions.**
3. **Other operators (+, -, etc.) are declared externally to the class.** This makes automatic conversions the same for left and right operands.

Inlines

1. **Use inlines sparingly.** ([Mey92] rule 33).
2. **Inline member function definitions appear outside the class declaration towards the end of the header file**. See the name-conflicts example in section 3.2.1. This makes the declaration easier to read and separates the definition from the interface (at the cost of duplicating the declaration syntax).

Templates

Friends

1. **Avoid friends** when writing C++ code. Seek their company otherwise. Friend functions are not as bad as friend classes. If you feel the need for a friend class, consider rearranging the class boundaries so that less communication needs to occur between the classes.

3.3 Comments

Good comments are absolutely critical for making code understandable and maintainable. Comments should be high-level descriptions. Aim as high as possible! Comments about purpose and functionality go in header files, as close to the code being commented as possible. Comments about implementation go in the `.cc` files. –although in some cases a quick clue about implementation *does* belong in the header file. For example, several classes might be related by a base-class, derived-classes, and handle-class structure, and this should be mentioned in the `.h` file.

3.3.1 Class declaration

Good comments are especially importance in header files, since these form the main interface in C++. The primary audience here is users: either end-users writing applications, or programmers writing code downstream. These people need to know what role this code plays in the overall code structure, and what each of the particular members and functions do. The comments in the declaration should make clarify

- what role this code plays in the overall scheme of things, and
- the purposes and uses of each functions and function argument.

Here's an example of an acceptably commented class declaration:

```
/// Multigrid solver on a level
/**
   Multigrid solver on a level.
   This class is to be considered internal
   to AMRSolver and not a part of the Chombo API.
 */
class LevelMG
{
public:

  ///
  bool isDefined() const;

  ///
  LevelMG();

  ///
  LevelMG(const DisjointBoxLayout& a_ba, ...

  ///
  void define(const DisjointBoxLayout& a_ba, ...

  /// Constructor for coarsened version of object.
  void define(const LevelMG& L, ...

  /// Constructor for coarsened version of object.
  LevelMG(LevelMG& L, ...

  ///
  ~LevelMG();

  ///
  void clear();

  ///
  /**: Invoke relaxation step. Default is pure MG V-cycle, suitable
     for use in multilevel solver application; otherwise, use approximate
     solver such as CG at bottom level. It is assumed that the problem has
     already been put in residual-correction form. In particular, only
```

```
     the homogeneous form of the physical and coarse-fine boundary
     conditions need be invoked.
  */
  void mgRelax(LevelData<FArrayBox> & a_soln, ...

  ///
  void setnumBottomGSRB(int a_numBottomGSRB)
  {m_numBottomGSRB = a_numBottomGSRB;}

  ///
  void setnumSmoothUp(int a_numSmoothUp)
  {m_numSmoothUp = a_numSmoothUp;}

  ///
  void setnumSmoothDown(int a_numSmoothDown)
  {m_numSmoothDown = a_numSmoothDown;}

  // this is a dangerous access function that should not generally be used.
  LevelOp* levelOpPtr();

  // this is another access function that is kinda bad
  LevelMG* lCoarsePtr();

protected:

  void setDefaultValues();
  void clearMemory();
  bool m_isDefined;

  //these are owned by levelmg
  //
  LevelData<FArrayBox>  m_resid;

  //
  LevelData<FArrayBox>  m_crseResid;

  ...

private:
  // correct fine on intersection with crse
  //should only be called internally because this
  //is not written for  general LDF's
  void crseCorrect(LevelData<FArrayBox>& a_fine, ...
  void operator=(const LevelMG& levmgin){};
  LevelMG(const LevelMG& levmgin) {};
```

```
};
```

3.3.2 Source code definitions

Architectural comments

There should be architectural-level comments in the .cpp files, describing the general structure of the classes and the broad implementation decisions. Leave, in comments, the biggest fattest, clues you can imagine to how you were thinking of the code when you wrote it. If the structure of the class follows a standard C++ idiom (the ref-counting handle, for example), say so. If the class is intimately related to another class (e.g. by derivation), say so, and in the central class's comments, write up a description of the global architecture of the related classes.

Here's an example of a decent architecture-comment:

```
// class Array<T>
// Simple templated array class, modeled after the C-language array.
// This implementation is simple in that there is no data-sharing or
// ref-counting between any Arrays. All copying is deep, each object is
// the sole owner of its member data. If we need a data-sharing/ref-cnt
// version, I'll implement it from this one.
```

Nitty-gritty algorithm comments

The lowest-level, line-by-line comments should help others understand (and yourself, in case you ever need to revisit code) exactly what you're doing, in the highest-level, most expressive English possible. Suggestion: write down what you'd say if you were explaining the algorithm to someone over the phone.

Chapter 4

Fortran Code

4.1 General Conventions

1. **Use Chombo Fortran**.
2. **All variables shall be defined.** Every subroutine should have an implicit none at the head (compiling with -u is not enough. Some compilers (i am told) ignore it).
3. **Reals shall be defined with the** `REAL_T` **macro.**
4. **All magic numbers shall be done with macros.** one, two, etc to avoid the real vs. double mess. This is perhaps less of an issue now that the vector crays no longer run our lives. Care should be taken when mixing literal constant and variables as arguments to Fortran intrinsics (e.g. SIGN) to make sure the precisions match. The Fortran standard does not allow mixing precisions in some cases.
5. **Code shall be indented according to the emacs Fortran standard**.

Bibliography

[Mey92] Scott Meyers. *Effective C++*. Addison-Wesley, 1992.

[Mur93] Robert B. Murray. *C++ Strategies and Tactics*. Addison-Wesley, 1993.