

# **SAND REPORT**

SAND2003-4169

Unlimited Release

Printed December 2003

## **Epetra Developers Coding Guidelines**

M. A. Heroux, P. M. Sexton

Prepared by  
Sandia National Laboratories  
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,  
a Lockheed Martin Company, for the United States Department of  
Energy under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.



Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from  
U.S. Department of Energy  
Office of Scientific and Technical Information  
P.O. Box 62  
Oak Ridge, TN 37831

Telephone: (865) 576-8401  
Facsimile: (865) 576-5728  
E-Mail: [reports@adonis.osti.gov](mailto:reports@adonis.osti.gov)  
Online ordering: <http://www.doe.gov/bridge>

Available to the public from  
U.S. Department of Commerce  
National Technical Information Service  
5285 Port Royal Rd  
Springfield, VA 22161

Telephone: (800) 553-6847  
Facsimile: (703) 605-6900  
E-Mail: [orders@ntis.fedworld.gov](mailto:orders@ntis.fedworld.gov)  
Online order: <http://www.ntis.gov/ordering.htm>



**SAND2003-4169**  
**Unlimited Release**  
**Printed December 2003**

# **Epetra Developers Coding Guidelines**

**Michael A. Heroux**

Computational Mathematics & Algorithms Department  
Sandia National Laboratories  
P.O. Box 5800  
Albuquerque, New Mexico 87185

**Paul M. Sexton**

Department of Computer Science  
Saint John's University  
Collegeville, Minnesota 56321

## **Abstract**

Epetra is a package of classes for the construction and use of serial and distributed parallel linear algebra objects [1]. It is one of the base packages in Trilinos [3]. This document describes guidelines for Epetra coding style. The issues discussed here go beyond correct C++ syntax to address issues that make code more readable and self-consistent. The guidelines presented here are intended to aid current and future development of Epetra specifically. They reflect design decisions that were made in the early development stages of Epetra. Some of the guidelines are contrary to more commonly used conventions, but we choose to continue these practices for the purposes of self-consistency. These guidelines are intended to be complimentary to policies established in the Trilinos Developers Guide [2].

**Acknowledgements:** These coding guidelines draw heavily from the NOX Coding Guidelines [4], which in turn are adapted largely from Programming in C++, Rules and Recommendations, by Mats Henricson and Erik Nyquist [5]. Valuable feedback was provided by James Willenbring and Alan Williams.

## Table of Contents

<b>1. Introduction.....</b>	<b>6</b>
<b>1.1 Guideline Categories.....</b>	<b>6</b>
<b>1.2 Grandfather Clause .....</b>	<b>7</b>
<b>2. Structure of Files.....</b>	<b>7</b>
<b>3. Naming Conventions.....</b>	<b>9</b>
<b>4. Formatting and Style .....</b>	<b>10</b>
<b>5. Coding Rules.....</b>	<b>15</b>
<b>6. Explicit Type Conversions .....</b>	<b>15</b>
<b>7. Error Handling.....</b>	<b>15</b>
<b>8. Output .....</b>	<b>16</b>
<b>9. Comments .....</b>	<b>16</b>
<b>References.....</b>	<b>18</b>

# 1. Introduction

This document describes guidelines for Epetra coding style. They are not intended to be rigid rules that must be mindlessly obeyed, or to enforce draconian notions of the one true way to program. Like any tool, their purpose is to help us be more productive. They are here because we have found them to be very useful and helpful. They are here because they give all Epetra code a cohesiveness and uniformity that makes it easier for us to read, write, and understand it. (One of the authors has personally been involved with refactoring some Epetra classes that he was not a designer or creator of. Guidelines such as these make an immense difference in the amount of time and energy required to understand a piece of code. Conversely, when things are inconsistent, it adds an additional obstacle to understanding what may already be a very complex program.)

## 1.1 Guideline Categories

Each guideline falls into one of three categories:

**VSR**

**Very Strongly Recommended** - Practices necessary for Epetra to build correctly, or considered of paramount importance.

**SR**

**Strongly Recommended** - Practices that are definitely a good thing, or that have proved to be valuable.

**R**

**Recommended** - Practices that are probably a good idea; also cases where there is no clear “best” way, and we are merely standardizing on one.

To determine which category a guideline belonged in, it was weighed along a two-dimensional scale. On one axis is the advantage it provides to the developers in terms of clarity, conciseness, etc. On the other axis is how badly things break if it is not followed.

Most of these rules can be broken, if there’s a good reason for it. However, that said, it is rare that there is a good enough reason, and extremely rare with the guidelines in the strongly recommended and very strongly recommended sections.

We realize that for many of the issues mentioned here, there is no one “best” way. (For example, indentation style.) But when all code follows the same formatting guidelines, that white space becomes a schema that conveys information about what that code does, and this information can be taken in while merely skimming a source file, without actually having to read through it all. The following quote is worth pondering, because we can draw direct parallels between the cows and Epetra developers, and between the cowpath and Epetra code.

“The living language is like a cowpath: it is the creation of the cows themselves, who, having created it, follow it or depart from it according to their whims or their needs. From daily use, the path undergoes change. A cow is under no obligation to stay in the narrow path she helped make,

following the contour of the land, but she often profits by staying with it and she would be handicapped if she didn't know where it was and where it led to.”<sup>1</sup>

## 1.2 Grandfather Clause

When these guidelines were developed, Epetra was already two years old and quite well established. As a result, there may be numerous instances in existing Epetra code where these guidelines are not followed correctly, only partially followed, or outright ignored. This is an unavoidable consequence of the order in which the guidelines and the classes came about. It would be a massive undertaking to search through and fix all the inconsistencies that may exist, and this was not considered to be a good use of developer time & energy. Thus, all Epetra code written prior to December 2003 is “grandfathered in”, and the presence of any inconsistencies should not be seen as a weakness of Epetra. Nor should it be viewed as an excuse for new code to not follow the guidelines as closely as possible.

# 2. Structure of Files

## Naming Conventions

VSR

- **2.1** C++ header files end in `.h` and source files end in `.cpp`

VSR

- **2.2** The name of the files should correspond to the name of the class they define, with an underscore after the word Epetra. For example, the definition of the class `Epetra_BlockMap` is in the file `Epetra_BlockMap.h`.

## Include File Structure

VSR

- **2.3** Every include file must contain a mechanism that prevents multiple inclusions of the file<sup>2</sup>. For example, the following should follow the header information for the `Epetra_BlockMap.h` header file.

```
#ifndef EPETRA_BLOCKMAP_H
#define EPETRA_BLOCKMAP_H

...body of include file goes here...

#endif /* EPETRA_BLOCKMAP_H */
```

---

<sup>1</sup> E.B. White. “The Living Language”, Writings From The New Yorker 1927 – 1976, HarperCollins, New York, 1990. p.143.

<sup>2</sup> For the rationale of this, see the section titled “Include Guards” in: Bjarne Stroustrup. The C++ Programming Language, Special Third Edition. Addison-Wesley, Reading, MA, 2000. §9.3.3, p. 216.

SR

- **2.4** Do not include system files (e.g., `iostream`) directly in your files. Instead, include `Epetra_ConfigDefs.h`. The goal is to better enable system portability since some machines have `<iostream>` and others have `<iostream.h>` and so on. Currently, we have the following system headers:

- `iostream`
- `iomanip`
- `stdio`
- `stdlib`
- `assert`
- `string`
- `math`

SR

- **2.5** MPI is not handled directly by `ConfigDefs`, and so an Epetra class that uses MPI must still include the MPI system files (typically `<mpi.h>`). It is handled indirectly, though, so all uses of MPI should be done within an `EPETRA_MPI` conditional.

```
#ifdef EPETRA_MPI
#include <mpi.h>
#include "Epetra_MpiComm.h"
#endif

#ifdef EPETRA_MPI

// code for MPI version goes here
Epetra_MpiComm Comm();

#else

// code for if MPI is not available (i.e. Serial build)
Epetra_SerialComm Comm();

#endif

// rest of code
int numproc = Comm.NumProc();
```

The `#else` part is not always necessary, but the class should function either way. Remember that the inclusion of MPI-related header files must also be inside of an `EPETRA_MPI` conditional.

```
#ifdef EPETRA_MPI
#include <mpi.h>
#include "Epetra_MpiComm.h"
#endif
```

SR

- **2.6** Definitions of classes that are only accessed via pointers (\*) or references (&) should be declared using forward declarations, and *not* by including the header files.

These are the cases when header files should be included in the header file:

- classes that are used as *base classes*,
- classes that are used as *member variables*,
- classes that appear as *return types* or as *argument types* in function/member function prototypes.

Note the distinction between a class being used as a member variable, and a pointer or reference to a class being used as a member variable. In the first case, it should be included. In the second, it should be forward-declared. That same distinction also applies to return types and argument types.

## 3. Naming Conventions

SR

SR

SR

SR

- **3.1** Class names should begin with an uppercase letter.
- **3.2** Class data members should end with an underscore (e.g., `int IndexBase_`). No other variable names should ever end with an underscore.
- **3.3** Do not use identifiers that begin with one or two underscores (`'_'` or `'__'`).
- **3.4** Accessor methods should have the same name as the attribute they access, sans the underscores.

```
int someVar_;
int someVar() {return(someVar_);};
```

R

- **3.5** In names (function, class, variable, etc) that consist of more than one word, the words are written together and each word that follows the first is begun with an uppercase letter. (e.g., `PointToElementList`).

R

- **3.6** Names should not include abbreviations that are not generally accepted.

```
int Original_LDA; // Ok. (LDA = Leading dimension across)
int New_RCD;     // BAD! (RCD = Received Column Data??
                  Residual Creation Decision??)
```

R

- **3.7** Choose variable names that suggest the usage. For example, consider the object being passed into this assignment operator as a right-hand side:

```
Epetra_Data& Epetra_Data::operator = (const Epetra_Data& Source) {
    ... // copy attributes of Source into this
    return(this);
}
```

R

- **3.8** The name of the parameter for a copy constructor or assignment operator should either be the name of the class, or `Source`.

```
Epetra_Data& operator = (const Epetra_Data& Source); // Yes
Epetra_Data& operator = (const Epetra_Data& Data); // Yes
Epetra_Data& operator = (const Epetra_Data& In); // No!
```

R

- **3.9** Variables used for loop counters should be named *i, j, k*, etc. in that order.

```
for (int i = 0; i < max1; i++) { // outermost loop uses i
    ...
    for (int j = userSize; j > 0; j--) { // next outermost uses j
        ...
    }
}
```

## 4. Formatting and Style

### Classes

SR

- **4.1** The public, protected, and private sections of a class are to be declared in that order (the public section is declared before the protected section which is declared before the private section).

SR

- **4.2** Friend class declarations should immediately precede the private section, to emphasize that they too can access those members. (It is legal C++ for them to be declared anywhere in the class).

R

- **4.3** The order functions are listed in the .cpp file should match the order they are listed in the class declaration in the .h file.

### Functions

SR

- **4.4** Always provide the *return type* of a function explicitly. The value being returned should be enclosed in parenthesis.

```
return i;        // No!
return(i);       // Yes
```

R

- **4.5** Functions with a return type of void should not use an “empty” return statement.

```
void foo() {
    ...
    return;    // No!!
}

void foo() {
    ...        // Yes
}
```

R

- **4.6** Always write the left parenthesis directly after a function name. There should be no spaces between the parenthesis and the expression inside of it.

```
void foo ();    // No!!
void foo();     // Better

void foo( int a ); // No!
void foo(int a); // Yes
```

R

- **4.7** When defining functions, the leading parenthesis and the first argument (if any) are to be written on the *same line* as the function name. If space permits, other arguments and the closing parenthesis may also be written on the same line as the function name. Otherwise, each additional argument is to be written on a separate line (with the closing parenthesis directly after the last argument).

```
EpetraClass::EpetraClass(int SomeReallyLongName,    // No!  
    double AnotherLongName, int YetAnotherLongName) {  
    ...  
}  
  
EpetraClass::EpetraClass(    // No!  
    int SomeReallyLongName,  
    double AnotherLongName,  
    int YetAnotherLongName) {  
    ...  
}  
  
EpetraClass::EpetraClass(int SomeReallyLongName,    // Yes  
    double AnotherLongName  
    int YetAnotherLongName) {  
    ...  
}  
  
EpetraClass::EpetraClass(int ShortName, int ShortName2) { // Also ok  
    ...  
}
```

R

- **4.8** Member definitions in constructors: Member definitions should be formatted as follows, each on their own line, with the colon preceding the first one, a comma following all but the last one, and the opening curly brace of the function body following the last one.

```
EpetraClass::EpetraClass(int foo)  
    : SomeMemberVar_(0),  
    SomeOtherVar_(foo + 1) {  
    ...  
}
```

SR

- **4.9** Whenever possible, we prefer member initialization to assignment in the body of the constructor.<sup>3</sup>

R

- **4.10** Functions that don't take any parameters should use an empty parameter list, and not say void. This makes it harder to identify, as you have to read it to know if it's void, or the name of a parameter. Leaving it empty makes it easier to spot while scanning.

```
void foo(void); // No!!  
void foo();    // Better
```

<sup>3</sup> See Item 12 in: Scott Meyers. Effective C++, Second Edition, Addison-Wesley, Reading, MA, 1998. pp. 52 – 57.

## Variable declarations

SR

- **4.11** Always define a pointer when you declare it. Either set it equal to an address in memory, or set it equal to zero. Do not declare a pointer and then leave it undefined, you never know if it will be accessed before you assign to it (This is not so much a problem now, when you're first writing the code, but later, when you or someone else is modifying it). This goes both for local variables, and for class members in constructors, both in the constructor body and in member initializations.

```
int* fooPtr = 0;           // Yes
int* fooPtr = new int[bar]; // Yes
int* fooPtr;              // No!!
```

SR

- **4.12** The characters '\*' and '&' should be written together with the types of variables instead of with the names of variables in order to emphasize that they are part of the type definition. Instead of saying that \*i is an int, say that i is an int\*.

```
int *i; // No!!
int* i; // Yes
```

R

- **4.13** Only one variable per line.

```
int i,j; // No!!
int i;   // Yes
int j;
```

This is mainly to avoid confusion resulting from mixing `int` and `int*` declarations.

```
int* a, b; // a is an int*, but b is an int.
```

A similar confusion can result from mixing declarations and definitions for multiple variables.

## Loops and conditionals: `if`, `for`, `while`, etc.

R

- **4.14** The opening parenthesis should be separated from the keyword by a space. This makes it easier to read, as the keyword and the parameters to it are separate typographic entities.

```
if(/*Something*/) { // No!
    /* Stuff */
}

if (/*Something*/) { // Yes
    /* Stuff */
}
```

R

- **4.15** There should be no spaces between the parenthesis and the expression inside of it. This goes for function parameters as well.

```

if ( i < 5 ) {    // No!
  /* Stuff */
}

if (i < 5) {      // Yes
  /* Stuff */
}

```

**R**

- **4.16** Use parenthesis to make code readable.

```

if (a == b && c < d || e == f) {           // No!
  /* Stuff */
}

if ((a == b) && (c < d) || (e == f)) {    // Yes
  /* Stuff */
}

```

**R**

- **4.17** The block of any if statement should always follow on a separate line.

```

if (/*Something*/) i++; // No!!

if (/*Something*/)      // Yes!
  i++;

```

**R**

- **4.18** Braces ("{}") that enclose a block should be aligned as follows, with the opening brace on the same line, separated by a single space. The closing brace should be indented to the same level as the surrounding statements.<sup>4</sup>

```

if (/*Something*/) { // Yes!
  i++;
  j++;
}

if (/*Something*/) // No!
{
  i++;
  j++;
}

if (/*Something*/) // No!
{
  i++;
  j++;
}

```

This applies to all instances where curly braces occur (functions, scoping blocks, and conditionals). The advantage of this is in “smart” text editors that will show you where

---

<sup>4</sup> Historical note: The indent style we use is called “K & R style”, after Kernighan and Ritchie, who formatted the examples in K&R C this way. The second example is known as “Allman style”, named after Eric Allman, who wrote a good portion of the BSD utilities. It is also sometimes known as “BSD style”. The third example is known as “GNU style”, since it is used in the GNU utilities.

the opening of a closing brace is. In XEmacs, for example, you can have it highlight the opening brace. If that brace is off the screen, it will show that line in a one-line buffer at the bottom. If the brace occurs on its own line, this tells us nothing. If the brace occurs at the end of the preceding line, we will know which function or conditional we are at the end of.

R

- **4.19** An else statement following an if should begin on the line following the if's closing brace.

```

if (a < b) {
  /* Stuff */
} else {           // No!
  /* Other Stuff */
}

if (a < b) {
  /* Stuff */
}
else {           // Yes
  /* Other Stuff */
}

```

SR

SR

R

- **4.20** Always include a default case in a switch statement, or in a sequence of if-elses.
- **4.21** “Fall-thru”-behavior is better avoided. If it is used, it should be documented as such.
- **4.22** If the body of a conditional is excluded, add a comment saying so. Otherwise future maintainers may erroneously assume that the following statement was meant to be associated with that conditional.

## Operators

SR

- **4.23** Operators should have a space on both sides of them. (The exception to this is the \* and & dereferencing operators). This makes it easier to distinguish which usage is intended.

```

int* a    // defining a pointer to int
a * b    // multiplying two variables
*a       // dereferencing a pointer

```

SR

SR

- **4.24** The greatest source of confusion is the \* and & operators, but these guidelines should be followed with all operators.
- **4.25** When dereferencing a pointer, there should be a space in front of the '\*' or '&', and no spaces between that and the variable name. A '\*' with a space on both sides should be interpreted as the multiplication operator.

```

classInst = *classPtr; // Yes!
classInst = * classPtr; // No!

```

SR

R

- **4.26** Do not use *spaces* around '.' or '->', or between unary operators and operands.
- **4.27** Always provide a space on both sides of = signs and all logical operators.

## Miscellaneous

R

R

- **4.28** Each statement should be on a separate line, however small it may appear.
- **4.29** Use the `c++` mode in GNU Emacs to format code. Adding the following line to your `.emacs` file will help:

```
(c-set-offset 'substatement-open 0)
```

## 5. Coding Rules

SR

- **5.1** A public member function should never return a non-const reference or pointer to member data.

SR

- **5.2** Constants should be defined using `const` or `enum`; never using `#define`.

SR

- **5.3** A switch statement should always contain a default branch that handles unexpected cases.

SR

- **5.4** Before deleting a pointer or pointer array, always check to make sure it's not zero. Always set the pointer to zero after you delete the object it's pointing to.

## 6. Explicit Type Conversions

SR

- **6.1** C-style casts should never be used. Use `static_cast`, `reinterpret_cast`, and `const_cast` instead.

“C-style casts should have been deprecated when the new-style casts were introduced. Programmers should seriously consider banning C-style casts from their own programs. Where explicit type conversion is necessary, `static_cast`, `reinterpret_cast`, `const_cast`, or a combination of these can do what a C-style cast can. The new-style casts should be preferred because they are more explicit and more visible.”<sup>5</sup>

R

- **6.2** `const_cast` should be avoided as much as possible. When you need to modify an object that is logically `const` but not bitwise `const`, use the `mutable` keyword instead.

R

- **6.3** In general, casting should be avoided, as the results are usually implementation-defined, and thus not portable.

## 7. Error Handling

SR

- **7.1** Always check return values of functions for errors.

SR

- **7.2** In general, try to recover from errors, either by catching an integer return value, or by enclosing code in a `try / catch` block.

SR

---

<sup>5</sup> Bjarne Stroustrup, The C++ Programming Language, Special Third Edition. Addison-Wesley, Reading, MA, 2000. §B.2.3, p. 819.

- **7.3** Epetra uses integer return values to notify the caller that an error occurred. The exception to this is functions that cannot return an `int` (such as constructors, `void` functions and functions returning an `int*`). They throw exceptions.
- **7.4** If you must throw an exception, use the `ReportError` method defined in `Epetra_Object.h`.
- **7.5** If your class/function throws exceptions, it must be mentioned in the class's documentation. This should include which functions may throw exceptions, what error codes they return, and what each of those codes mean. For example:

Epetra\_BlockMap constructors will throw an exception of an error occurs. These exceptions will always be negative integer values as follows:

1. -1 NumGlobalElements < -1. Should be >= -1 (Should be >= 0 for first BlockMap constructor).
2. -2 NumMyElements < 0. Should be >= 0.
3. -3 ElementSize <= 0. Should be > 0.
4. -4 Invalid NumGlobalElements. Should equal sum of MyGlobalElements, or set to -1 to compute automatically.
5. -5 Minimum global element index is less than index base.
6. -99 Internal Epetra\_BlockMap error. Contact developer.

## 8. Output

- **8.1** Many of the Epetra base classes have a `Print()` function defined. This prints out general information about the object and its state and variables. (For example, an `Epetra_Comm` would print out information about the number of processors, and the local Processor ID.) Each implementation of that abstract base class implements the `Print` function by calling an overloaded `ostream <<` operator. It is recommended that users call the `Print` function, rather than trying to call the overloaded `<<` operator directly.
- **8.2** Epetra does not have its own output mechanism, other than that described above. All output is done using standard `ostream` calls.

## 9. Comments

- **9.1** We use Doxygen for the comments [6]. However, a tutorial on how to use Doxygen is beyond the scope of this document.
- **9.2** Always document every class, and every function in the header file.
- **9.3** Private / protected functions won't be used by the user, but should still be documented just like public functions. Putting a comment in the source code is not enough; there should be a Doxygen entry as well. This is invaluable to future maintainers (including yourself).
- **9.4** If a file doesn't contain the header or source for a class, like `Epetra_DataAccess.h`, it should still have a file-level Doxygen comment explaining what's in that file. Functions, enums, etc. should also have comments given.

- **9.5** Any functions that throw exceptions should be documented, listing what will cause the exception, and what error code and message will be given.

# References

- [1] Epetra Home Page: <http://software.sandia.gov/trilinos/packages/epetra>, 8 December 2003.
- [2] M. Heroux, J. Willenbring and R. Heaphy, *The Trilinos Developers Guide, Version 1.0*, Sandia National Laboratories, SAND2003-1898, August 2003.
- [3] Trilinos Home Page: <http://software.sandia.gov/trilinos>, 8 December 2003.
- [4] NOX Developer's Coding Guidelines: <http://software.sandia.gov/nox/coding.html>, 8 December 2003.
- [5] Mats Henricson and Erik Nyquist. "Programming in C++, Rules and Recommendations". <http://www.doc.ic.ac.uk/lab/cplus/c++.rules/>, 8 December 2003.
- [6] Doxygen Home Page: <http://www.doxygen.org>, 8 December 2003.