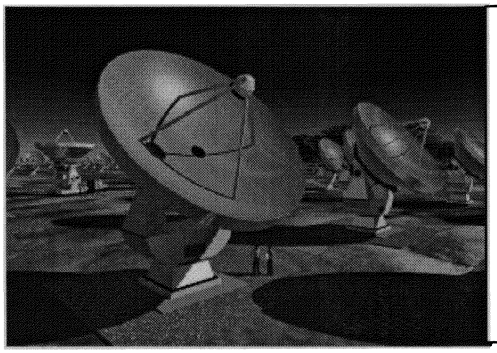
ALMA Computing Memo No. 10



Atacama Large Millimeter Array	ALMA-SW-0010 Revision: 5 2001-08-10 *Software Standard* Alan Bridger

C++ Coding Standards

Software Standard

Alan Bridger
UK Astronomy Technology Centre
Jim Pisano
National Radio Astronomy Observatory

Keywords: programming standards, C++, language

Author Signature:	Date:
Approved by: Brian Glendenning, Gianni Raffi	Signature:
Institute: NRAO, ESO	Date:
Released by:	Signature:
Institute:	Date:

Change Record

REVISION	DATE	AUTHOR	SECTIONS/PAGES AFFECTED
	REMARKS		
1	2000-10-16	Alan Bridger	All
	Reformatted, tidied, minor changes, DOC++ added (from JP original)		
2	2000-10-30	Alan Bridger	Section 6, appendix I
	Up to date version of DOC++ used.		
3	2001-02-28	Alan Bridger	All
	Responded to review comments, assigned doc. #.		
4	2001-06-11	Alan Bridger	
	Namespace naming, exception guideline, copyright/license mods.		
5	2001-08-8	Michele Zamparelli	
	substituted DOC++ with doxygen		

Table of Contents

1 C++ Coding Standards

1.1 Scope

This document describes the C++ coding guidelines agreed upon by the software engineering team. These guidelines are expected to be adhered to in all future C++ software development. Existing code will not be required to be rewritten to adhere to these guidelines, but it is suggested that reasonable efforts are made to adhere to the new guidelines when working on existing code.

1.2 Introduction

This document was written with the following goals in mind:

1. Code should be robust and error free.
2. Code should be easy to use and understand.
3. Code should be easy to maintain.

Each section of this document contains recommended **standards** in that area, along with a further list of **guidelines**. A **standard** is required to be followed in all future development. If a programmer must violate a **standard**, then the reason for doing so must be strong and clearly documented via code comments. **Guidelines** are just that. Individual programmers are encouraged to program in a style which is consistent with the rest of the programming team, but differing code styles are acceptable.

These standards recognize that individual programmers have the right to make judgments about how best to achieve the goal of code clarity. When working on others' code, please respect and continue existing styles which, although they may differ from your own, meet these guidelines. This in turn leads to what is perhaps the most important aspect of coding style: consistency. Try, as much as possible, to use the same rules throughout the entire program or module.

2 Class Declarations

Standards

- Robust, error-free, easy to use, understand, and maintain class declarations are the single most critical element of programming style, see [5], item 18.
- Class declarations should be platform invariant to the greatest extent possible. Any compiler conditional directives should be minimal and documented.
- Use include guards in header files to prevent multiple inclusion.
- Within each section, member functions and member data should not be interspersed.

- The default constructor, copy constructor, assignment operator, and destructor should be either explicitly declared or made inaccessible and undefined rather than relying on the compiler-generated defaults. See [5], item 27.

2.1 Organization

Guidelines

- There should be at most one **public**, one **protected** and one **private** section in the class declaration. They should be ordered so that the **public** section comes first, then the **protected** section, and lastly the **private** section.

- Member functions and data should be listed logically. For example, all constructors should be grouped together, all event handling routines may be declared together, as may the routines which access member data. Each logical group of functions should have a common comment above the group explaining why they are grouped together.

- In general there will be one class declaration per header file. In some cases, smaller related classes may be grouped into one header file.

2.2 Contents and Structure

Standards

- Class data members must always be `private`. If access to them is required then this must be provided through public or protected member functions. See [5], item 20.

- Use `const` whenever possible for function, reference and pointer parameters, and data member declarations, except where non-constness is required. Use the `mutable` keyword to as needed for data which is logically but not physically const (for example, read buffers implemented for efficiency).

- A class's declaration must expose the logical usage of the class, and protect or hide its implementation details. The capacity to make the greatest degree of enhancement and modification with the least change to class declaration is highly desirable for effective maintenance, especially for class interfaces exposed in shared libraries.

Guidelines:

- Forward declarations should be employed when safely applicable to minimize #include dependencies and reduce compilation time. Forward declarations may be applied to avoid including declarations of objects that are used solely by pointer/reference or as function return values; they may not be used if the object serves as a base class, or as a non-pointer/reference class member or parameter. See [5], item 34.

- Implementation details of complex class declarations should be moved to the class definition to reduce #include dependencies and to enhance the ability to modify class implementation without changing the class declaration.

- Organize class member functions in your source code in the same order they are declared in the class interface.

- For classes requiring a significant amount of source code in their implementation, identify member function groupings by their accessibility scope (i.e. "public member functions", "private member functions", "static member functions", etc.).

3 Naming Conventions

Standards

- Clear and informative names are one of the best tools for creating easily understandable code. The name of any identifier should succinctly describe the purpose of that identifier.

- Avoid abstract names (in a global context) that are likely to be reused by other parts of the system.

Guidelines

- It is recommended that names be a mixture of upper and lowercase letters to delineate individual words or separated by underscores . Use descriptive names and avoid abbreviations except where the abbreviation is an industry or project standard.

- See [5], § 9 for a more detailed set of guidelines.

3.1 Class Names

Standards

- Class names must identify the type of the object they represent. (e.g. "Message" or "OutputDevice").

Guidelines

- Class names will consist of nouns or noun combinations. Derived class names should be suffixed by base class names (e.g. "ClockOutputDevice" or "EditorCWin").

- The capitalization rule for class names should be all words in the name capitalized, e.g. "ClassName".

3.2 Namespaces

Standards

- Namespace names must be related to the domain they delimit.

- Namespace names must identify the namespace uniquely.

Guidelines

- The capitalization rule for Namespace names should be all words in the name capitalized, e.g. "MySpace". No underscores are allowed.

3.3 Function Names

Standards

- Function names must identify the action performed or the information provided by the function.

Guidelines

- Function names will generally begin with a verb.
- The capitalization rule for function names should be all words in the name capitalized, except for the first, e.g. "**functionName**".

3.4 Variable Names

Standards

- The type and purpose of each variable should be evident within the code in which it is used. E.g. the reader will expect `counter` to be an `int`, `motorSet` might be a `BOOLEAN` or an array representing a set – context will usually clarify this. And while the type of `sessionId` might not be obvious it will be expected to be an identifier or handle that labels some sort of session. Class member data must be easily distinguishable from local data in a consistent manner.
- Names should be formed from composite words, delimited by upper case letters, with no underscores allowed (except for appending pointer identifiers etc.) Underscores should be used as delimiters in macro names.

Guidelines

- Variable names should be short, but meaningful.
- Local variables should be named with the content of the variable. Each word except the first one is capitalised. For example, `counter` and `sessionId` are acceptable variable names.
- Global variables should be similarly named except that each word is capitalised. For example, `ExposureTime` is acceptable.
- Pointer variables should be identified, preferably by appending "`_p`", for example:

```
struct s * meaningfulName_p
```

- Identify class member variables by appending "_m", where "m" stands for "member." Valid member variable names would include "fileName_m" and "recordCount_m".

- The capitalization rule for variable names should be all words in the name capitalized except for the first, e.g. "**variableName**".

3.5 Other Names

Standard

- Macros, enumeration constants and global constant and typedef names should be stylistically distinguished from class, function, and variable names.

- Types (struct types, etc.) should be given meaningful names and begin with an uppercase letter in order to distinguish them from variables.

Guideline

- Macros, enumeration constants and global constant and global typedef names should be in all uppercase with individual words separated by underscores, e.g. DATA_VALID, const int MIN_WIDTH = 1;

3.6 File Names

Standards

- Header file names should have the extension "**.h**".

- C++ Implementation (source) file names should have the extension "**.cpp**".

- File names should contain only alphanumeric characters, "_" (underscore), "+" (plus sign), "-" (minus sign), or "." (period). Meta-characters and spaces must be avoided. Also file names that only vary by case are not permitted.

- When used with the ESO CMM filenames should contain the module name as a prefix.

Guideline

- In general, file names should declare the contained class name.

4 Code Commenting

Standards

- All files, both header and source, must begin with the standard header information, which contains the file identification information, a one line description of the class, and the copyright notice. See the example header files in Appendix A. Different copyright statements may be required for different sites.
- The header may also contain a longer description of the purpose of the class and any other pertinent information about the class.
- A change log for each module must be maintained in a manner appropriate to the development environment. Exactly how this is done is still to be determined by the development environment chosen.
- Comments intended for documentation should use the *doxygen* style. A template for the ALMA project may be found in Appendix A.
- Block style and in-line comments are both acceptable.
- Block style comments should be preceded by an empty line and have the same indentation as the section of code to which they refer. Block style comments should appear at the beginning of the relevant segment of code. C++ style comments ("//") are preferred.

Block Style:

```
//
// ..............
// ..................
// ......................
//
```

- Brief comments on the same line as the statement that they describe are appropriate for the in-line commenting style. There should be at least 4 spaces between the code and the start of the comment.

In-Line Comments:

```
..........          // .........
..........          // .........
..........          // .........
```

- Use in-line comments to document variable usage and other small comments. Block style comments are preferable for describing computation processes and program flow.

Guidelines

- Use the **Id** RCS identifier for all RCS-based source-code control tools. The **Id** identifier expands to **$Id: filename revision date time author state $.** When used with ESO's CMM the identifier must be preceded by **@(#)**.

- Use a "ToDo" section in the header comments to indicate those items that remain to be done.

4.1 Header Files

The primary goal of interface comments is to define the class and its members at a level appropriate for a new user of the class. Interface comments should be as concise as possible to meet this goal.

Standards

- Documentation comments will allow for **doxygen** extraction to provide a general understanding of the class and its interface. See Section 6.

- Interface commenting must be complete and up-to-date at all times during a class' lifetime.

- Each member data element must be described.

- Each member function must be described. In addition to specifying what the function does, this description must include:

  a) For copy constructors and assignment operators, whether or not copy or reference semantics are used.

  b) What the parameters are, how they are used, and any preconditions that must be established for the parameters. A variable name, in addition to its type, must be used to identify the parameter's purpose.

  c) What the function is going to do to the parameters and to the object itself.

  d) What the results/return values are in all the different cases possible for the function.

  e) What exceptions the function might throw.

Guideline

- The preferred format for member data element comments is **doxygen** style comments (///) above at the same indentation level of the code that is being discussed. C++ style comments (//) may be used when documentation is not required.

4.2 Source Files

Standards

- All comments in source code files must be up-to-date at all times during that code's lifetime.

- Code comments should be used to give an English language synopsis of a section of code, to outline steps of an algorithm, or to clarify a piece of code when it is not immediately obvious what was done or why it was done. In no case should the code comments just parrot the code.

Guidelines

- Code comments which apply to a block of code (either a loop or branch construct, or a grouping of statements), should be immediately above the block and indented to the same level as the code.

- Code comments which apply to a single statement may be immediately above or to the right of the statement. If above the statement, the comment should be indented to the same level as the code. If to the right of the statement, sufficient white space should be used to separate the comment from the code.

- Use a line of dashes '—' to visually block off member function definitions in the source file. This allows easy identification of where the functions start.

5 Object Implementation Guidelines

This section provides some basic implementation guidelines for C++. It does not provide a complete nor exhaustive list of guidelines that ensure correct C++ usage. There are many excellent references which can provide this information some of which are listed in section 7 ([3] and [5]), Meyer's book is especially insightful.

5.1 Structure

Standards

- In general, hide the implementation details from the interface. This encompasses issues including data member accessibility (avoid public), implementation of assignment operators and copy constructors, and judicious use of inheritance and virtual functions. See [5] items 20, 11-17, and 35-43.

- Minimize the use of the global name space.

- For software intended for workstations make use of namespaces to partition the global name space. For software intended for LCUs (and for common software that can be used on LCUs or workstations) namespace support cannot be guaranteed and so namespaces should not be used.

- When namespaces are not used, **structs** can be used to break up the global name space. Global scope functions, data, and even types should be kept to a minimum, and these identifiers linked to the most logically-related class. Functions used within a class should be member functions of that class rather than hidden global scope functions. Enumerated types used specifically for one class should be declared within that class. See [5], items 28, 32 and 47. Also of note is a macro definition in ACE that simulates namespaces.

Guidelines

- Global static objects should be avoided. If they are used, care should be used when declaring global static objects as they cause initialization ordering problems and cause problems in a multi-threaded environment. An instantiator class which counts references to an object should be implemented. See [4], topic 4.04 or [5], item 47.

- Project-wide header files which declare, include and define everything useful and shared for an entire project simplifies the project organization. This process must be somewhat controlled as it can lead to large includes and slow the build process dramatically. As the project matures, this project-wide header can be pared down with forward declarations replacing include files.

- If used, exceptions should be specified in the signature of the class method declaration and definition. This should include any exceptions that might be thrown directly or indirectly by the method.

- Create your own exception heirarchies reflecting the domain and define relevant exception classes derived from the standard exception class. See [10]. E.g.

```
class OutputDeviceException : public exception {..};
class OutputDevice : public Device
{
   .
   .
   .
};
```

- Users of ACE TAO on vxWorks may still be constrained to using the ACE exception emulation and should follow the ACE guidelines.

5.2 Robustness

Function definitions must -- to the fullest extent possible -- trap invalid or undefined use of the class.

Standards

- Declare `const` variables instead of using #defines for simple constants. The compiler will provide a level of type checking when the constants are used.

- Initialize all member data and local variables. All pointers should be initialized to appropriate values or NULL.

- Use `ASSERT()`'s or comparable debugging macros liberally to trap potential programming errors (in your class' code or in the calling function's code). Validate all parameters passed to any `public`, `protected`, or even `private` function. Verify all assumptions about the environment in which each function is called. Verify intermediate values calculated by your algorithms.

- Handle all potentially-invalid parameters or environmental conditions in a graceful, consistent, and documented manner. Remember that the final version of your class should be compiled without the `ASSERT()`'s described above, so it will need additional code to trap potentially-fatal errors.

Guidelines

- Keep local variables as local as possible. A counter which is used only in one `for` loop should be initialized in that loop, rather than at the top of the function. Keep in mind however, that any variables declared inside nested loops will be repeatedly constructed and destructed. In cases where construction or destruction are expensive, it may be preferable to declare the variable outside the loop.

- Pass parameters by reference rather than by pointer unless pointer manipulation or a null pointer is a possibility.

- Read "Writing Solid Code" [6].

5.3 Code Style

Standard

- Use consistent nested indentation throughout your code. When working in existing code, respect the indentation style currently in use. At a minimum, indentation should be consistent within a function or a class declaration.

Guidelines

- Nested indentation should be 4 characters at each level.

- Avoid very long functions which may be difficult to comprehend and maintain. If a function becomes too long, break it into logical chunks and put each chunk into a function of its own. A function that is more than 100 lines, including comments and white space, is generally considered to be too long. Some authors recommend that a function should fit on one screen of your text editor.

- Files longer than 1000 lines should be avoided.

- Use braces liberally. It is a good idea to put braces around any branch or loop code even if it is a single statement, because sooner or later someone will want to add another statement to the loop and forget to add the brace.

- Use parentheses liberally. They do not add code, but they can help avoid precedence or ordering mistakes, and can help readability.

- Use white space liberally to make code more readable by putting space before and after operators and between the sections of a complex statement.

- Line length should not exceed 80 characters as telnet interfaces are sometimes used to deal with code.

5.4 Organization

Guidelines

- Tricky C or C++ syntax should be avoided; clarity should be emphasized, rather than emphasizing intricacy or cleverness or brute conciseness. Conciseness is an

admirable goal, but it should not be allowed to interfere with understandability or maintainability.

- Optimize code only after performance measurement shows where the time is going. It is easy to incorrectly guess what areas of code are the bottlenecks. See [7].

- White space should be used to group functions, and to group steps of algorithms within functions.

5.5 Debugging

Guideline

- Use tracing statements at critical points in your code, or to follow execution paths. Include the module and function name in each tracing statement you use.

6 Doxygen Use

The ALMA standard code documentation tool is doxygen. In this section a suggested standard doxygen usage is presented. It is not intended that this be a doxygen primer - the documentation for doxygen is easily available (see [8]) and is much better for that. Note that the latest version of doxygen is 1.2.8.1

In very brief terms doxygen uses special comment delimiters (/**...*/ or ///... and //@{...//@}) to allow the programmer to add his or her own documentation comments. In addition doxygen constructs class hierarchies.

In addition special tokens, similar to Javadoc ones, allow the specification of particular attributes such as author, version, etc.

The following are the recommended standards for using doxygen for ALMA software. A suggested example ALMA usage of doxygen may be found in the appendices, including examples of each of the following.

- Use @Include to include files for related classes..
- Use @see (where appropriate) for class documentation. Author and version are stored in the code repository.
- Use @exception, @return, @pre, @post, and @param (where appropriate) for C++ methods. The class Bar has a constructor method with examples.
- Use a standard configuration file to store doxygen options.
- Document private members. In general it seems better to provide more rather than less documentation about the class.
- Use upwards arrows in the class graphs to conform with accepted generalisation convention.

A doxygen configuration file is included in the example. It should be noted that this is a suggested doxygen usage and that no doubt it will solicit comments. The authors feel that

a separate document on doxygen usage for IDL, C, C++ and Java might be more appropriate.

7 References

1. Taligent's Guide to Designing Programs, Taligent Press, 1994.
2. Large Scale C++ Software Design, Lakos, J., Addison-Wesley Publishing Company, 1996.
3. C++ Programming Guidelines, Plum, T., Saks, D., Plum Hall, 1991.
4. Java Code Conventions, 1997, Sun Microsystems.
5. Effective C++: 50 Specific Ways to Improve Your Programs and Designs, Meyers, S., Addison-Wesley, 1992.
6. Writing Solid Code, Maguire, S., Microsoft Press, 1993.
7. More Effective C++: 35 New Ways to Improve Your Programs and Designs, Meyers, S., Addison-Wesley, 1996.
8. Doxygen, van Heesh, D., http://www.doxygen.org, 2000.
9. cxx2html - C++ to HTML Converter, Schiebel, D., http://aips2.aoc.nrao.edu/docs/html/cxx2html.html, 1995.
10. C++ Coding Standards, The Core Linux Consortium, http://corelinux.sourceforge.net/cppstnd/cppstnd.html, 2000.

Appendix A. Doxygen Example

In this appendix are three example header files for three classes describing a very simple class tree and one for a global function. The templates show how doxygen comments can be inserted into source code. Note that example copyright and license statements are included here. The ALMA project may agree on a standard license statement, but each partner may be required (by funding bodies) to insert their own copyright statement.

Foo.h

```
Foo.h
#ifndef FOO_H
#define FOO_H

// @(#) $Id$
//
// [Insert copyright statement appropriate to author(s).]
//
// Produced for the ALMA project
//
// This library is free software; you can redistribute it
```

```
// and/or modify it under the terms of the GNU Library
// General Public License as published by the Free Software
// Foundation; either version 2 of the License, or (at
// your option) any later version.
//
// This library is distributed in the hope that it will be
// useful, but WITHOUT ANY WARRANTY; without even the
// implied warranty of MERCHANTABILITY FITNESS FOR A
// PARTICULAR PURPOSE.  See the GNU Library General Public
// License for more details.
//
// You should have received a copy of the GNU Library
// General Public License along with this library; if not,
// write to the Free Software Foundation, Inc., 675
// Massachusetts Ave, Cambridge, MA 02139, USA.
// Correspondence concerning ALMA should be addressed as
// follows:
//
//        Internet email: alma-sw-admin@nrao.edu


/// Foo base class, fundamental piece of code
/**
 *<P>
 *  A sample Foo device.
 *<P>
 * A Foo object doesn't do much here, but hold a count to get & set.
 *  It is used by FooInstantitor to blah, blah.
 *
 * <pre>
 * // Construct a foo object
 *    Foo myFoo;
 *    myFoo.setCount(0);
 *    int fooCount = myFoo.getCount();
 * </pre>
 *
 *  @see         Bar
 *
 */

class  Foo
{

public:
  /**
   *  constructor
   *  A more detailed description of the constructur
   *  and its peculiarities
   */
    Foo();

  /**
   *  constructor with param
   *  A more detailed description of this constructor
   *  and why it differs from the previous one
   */
    Foo( const Foo &);

  /// destructor, a simple one

  ~Foo();
```

```
  /// Overload assignment
  Foo &operator =( const Foo &);

  /// A get function
  int getCount() const;

  /**
   * A set function
   * the peculiarity of this function
   * @param new value the counter should be reset to
   */
    void setCount( const int _newValue);

  /// a public member function showing links to argument
  /// and type classes
  const Bar&    getBar( const Intermediate& c ) const ;

protected:
  /// a protected member variable
  double doubleCount;

private:
  /**
   * private members only  show up if your doxygen configuration
   * file is set up accordingly
   */
  int m_currentCount;

};
#endif
```

Intermediate.h

```
#ifndef INTERMEDIATE_H
#define INTERMEDIATE_H

// @(#) $Id$
//
// [Insert here the same copyright and license statement as presented
above]

//@Include: Foo.h

/**
 *  Just to make the class graph look more a little more
 *  interesting. Here we show multiple inheritance from one
 *  docified class and a nondocified one.
 *
 * @see         Foo, NotDocified
 */

class Intermediate : public Foo, public NotDocified
{
};
```

```
#endif
```

Bar.h

```
#ifndef BAR_H
#define BAR_H
// @(#) $Id$
//
// [Insert copyright and license statement as presented above]
//
//


/**
 *  A derived class.
 *  Here we show inheritance from a single docified class.
 *  This example shows how to structure the members of a
 *  class, if desired.
 *
 *  @include: Function.h
 *  @include: Intermediate.h
 *  @see            Foo
 */

class   Bar : protected Intermediate
{
public:
    /**@name    Parameters  */

    //@{

    /// the first parameter
    double  a ;

    /// a second parameter
    int     b ;
    //@}

    // constructor
    /**
     * This constructor takes two arguments and serves as
     * a suggested example of how ALMA methods should be
     * documented. It demonstrates documenting parameters
     * exceptions, return values, and pre- and post-
     * conditions.
     *
     *     @param  a this is good for many things
     *     @param  b this is good for nothing
     *     @return A Bar, whatever that is.
     *     @exception      None
     *     @pre   None required
     *     @post  Whatever must happen next
     */

    Bar( double a, int b ) ;

    /// destructor
    ~Bar() ;

    /**@name    These methods belong together */
```

```
    //@{

  /// Calvin's transmogriphier must be loaded
  void transmogriphierLoad(int profile);

  /// the transmogriphier must be unloaded after usage
  int transmogriphierUnload(void);

  /// this activates the transmogriphier
  void transmogriphierStart(void);

    //@}
};
#endif
```

function.h

```
#ifndef FUNCTION_H
#define FUNCTION_H

// @(#) $Id$
//
// [Insert copyright and license statement as presented above]
//
//

//* @file This file tag is strictly necessary for doxygen */

/** A simple global function.
 *
 *  This is an example for how to document global scope
 *  functions. You'll notice, that there is no technical
 *  difference to documenting member functions. The same
 *  applies to variables or macros.
 *
 *  @param  c   reference to input data object
 *  @return Some integer
 *  @see    Bar
 */

int function( const Bar& c ) ;
#endif
```

Appendix B. Example HTML Output

Since doxygen produces HTML output that has embedded JavaScript code it is easiest here to simply provide a URL to the example set out above. The doxygen output may be found at http://www.eso.org/~mzampare/alma/doxygen/.

---

[1] For the class hierarchies. This may be overridden for a simpler, plain HTML representation.

This output was produced by running doxygen on the above with the command

```
doxygen alma-doxy
```

run in the directory where the header files are, and using some simple HTML files for customised headers and footers. The doxygen options were specified in the alma-doxy configuration file. The ALMA project should adopt a standard configuration file.

The customised headers are examples. The ALMA project may wish to adopt standard headers, but it may also be appropriate for each package to provide its own standard headers. The example banner/footer is a suggested ALMA standard.