EGO European Gravitational Observatory

C++ Coding Standards and Guidelines for EGO

Author	EGO-SW
Date	2006-01-27
Version	v1r1
Document Nr.	EGO-SPE-OPE-75

EGO - European Gravitational Observatory
Traversa H di Via Macerata - Santo Stefano a Macerata, -56021 Cascina, Italia.
Secretariat: Telephone (+39) 050 752 521 FAX (+39) 050 752 550

Change Record

Version	Date	Section Affected	Reason / Remarks
v1r1	2006-01-27	All	Comments from MEV incorporated
v1r0	2005-06-17	All	Format reviewed and codifier added
0.3	2004-08-30	3.2	Example of namespace usage
0.2	2004-07-10	3.7	Added p as prefix to pointer naming convention.
0.1	2004-06-08	All	First Draft

1 Table of Contents

1 Table of Contents 3
- 1.1 Purpose 4
- 1.2 Scope 4
- 1.3 Applicable Documents 4
- 1.4 Reference Documents 4
- 1.5 Abbreviations and Acronyms 4
- 1.6 Glossary 4

2 Class Declarations 5
- 2.1 Class layout 5

3 Naming Conventions 6
- 3.1 Class Names 6
- 3.2 Namespaces 6
- 3.3 Function/Method Names 7
- 3.4 Function/Method Arguments 7
- 3.5 Class attribute names 7
- 3.6 Variables on the stack 8
- 3.7 Pointer variables 8
- 3.8 Reference variables and functions returning references 8
- 3.9 Global Variables 9
- 3.10 Global Constants, Typedefs, Enumerations 9
- 3.11 C Functions 10
- 3.12 File names 10

4 Code formatting 11
- 4.1 Commenting 11
- 4.2 Indentation and Alignment 12
- 4.3 Switch Statements 13

5 General comments 13

6 Example 14

1.1 Purpose

This document provides a minimal set of standards and guidelines for development in C++. It is not expected to change already existing software but it is expected to adhere to this standard for any future software developments.

Adhering to software standards will result in consistency in code, better quality, easier maintenance and more productive development, especially when multiple developers are involved.

1.2 Scope

This document describes general guidelines for software development at EGO as well as standards with respect to naming of files, variables, classes etc., class declarations as well as comments.

1.3 Applicable Documents

1.4 Reference Documents

1.5 Abbreviations and Acronyms

1.6 Glossary

2 Class Declarations

2.1 Class layout

- There should be at most one **public**, **protected** and **private** section in a class declared in the above order.

- Member functions should be group together.

- Data members and member functions should be grouped separate.

- 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.

- Use ‘include’ guards in header files to prevent multiple inclusion.

- Class data members must always be **private**. If access to them is required then this must be provided through public or protected member functions.

- 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.

3 Naming Conventions

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

Use descriptive names and avoid abbreviations except where the abbreviation is an industry or project standard.

It is recommended that names be a mixture of upper and lowercase letters to delineate individual words. (e.g. IsMaxOffset() or ClockOutputDevice)

3.1 Class Names

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

- Class names should be prefixed with the package/library to which they belong. (e.g. LFKalmanFilter where LF is the prefix for Linear Filtering package)

- Class names of derived classes should be suffixed with the base class. (e.g. ClockOutputDevice where the class ClockOutputDevice is derived from the base class OutputDevice)

3.2 Namespaces

- Namespaces must be related to the domain they delimit.
- Namespaces must identify the namespace uniquely.
- Use capitalization for every word in the namespace.

In general, using a namespace in a source file (.cpp) one should make use of the **using** directive, thus writing: using namespace std; at the top of the source file.

When referring to variables in a header file, the scope resolution operator should be used ::, thus writing namespaces::variable/function.

3.3 Function/Method Names

- Function or method names should identify the action performed or the information provided by the function

- It will generally begin with a verb

- Capitalization is the same for Class names namely every word in the name must begin with a capital letter. (e.g. IsMaxLimit())

3.4 Function/Method Arguments

- Where more than one argument is passed to a function/method, the arguments should be listed one per line to improve readability.

Example:

```
int  StartYourEngine(
                  Engine& rSomeEngine,
                  Engine& rAnotherEngine);
```

3.5 Class attribute names

- Class attribute names should be prepended with the letter m to indicate a member or ptr to indicate a pointer to a member.

- After that, use capitalization as for class names.

Example:

```
int                 mVarAbc;
int                 mErrorNumber;
String*             mName_ptr;
```

3.6 Variables on the stack

- Any other variables can be defined with the use of underscores ("_") and all lower case letters.
- Local variable declarations should be located at the top of the function, except in cases where doing so limits the scope of the variable (e.g. inside a loop).

Example:

```
NameOneTwo::HandleError(int errorNumber)
{
    int               error = OsErr();
    Time              time_of_error;
    ErrorProcessor    error_processor;

}
```

3.7 Pointer variables

- Pointer variables should be prefixed with the letter "p" or appended with "_ptr"

- Place the * close to the type and not the variable name

Example:

```
String* pName = new String;
String* Name_ptr = new String;
```

3.8 Reference variables and functions returning references

- References should be prepended with the letter "r"

Example:

```
class Test
{
  public:
     void DoSomething(StatusInfo& rStatus);

     StatusInfo&        rStatus();
     const StatusInfo&  Status() const;

  private:
     StatusInfo&      mrStatus;
     StatusInfo*      mStatus_ptr;
}
```

3.9 Global Variables

- Global variables should be prefixed with the letter “g”.

Example:

```
ErrorLogger  gLog
ErrorLogger* gLog_ptr
```

3.10 Global Constants, Typedefs, Enumerations

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

- Global constants, Enumeration Labels should be all capital letters separated with “_” and should be prepended with the software package prefix.

Example:

```
const int PACKAGE_GLOBAL_CONSTANT = 5;
```

3.11 C Functions

- In a C++ project there should be very few C functions
- Define C functions according to the GNU convention of all lower case letters delimitated with “_”.

Example:

```
int some_function()
{

}
```

3.12File names

- 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.
- Filenames should generally contain the package name as a prefix.
- In general, file names should declare the contained class name.

Example:

```
LFKalmanFilter.h
LFKalmanFilter.cpp
```

4 Code formatting

Good formatted coded aides in readability and understandability, which will eventually contribute to maintainability, thereby a necessary thing but a sensitive topic.

Code formatting applies to the way comments are made inside source files, indentation, and the use of brackets for code blocks.

4.1 Commenting

- 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.

- 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. Currently it is sufficient to provide a package-wide changelog in the root of the package.

- 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.

- Whenever possible automatic generation of documentation via doxygen should be used.

Example:

```
//
// comment
// comment
//
```

- 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.

Example:

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

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

4.2 Indentation and Alignment

Indentation is a very important for aiding readability.

- Rather uses spaces than tabs, as the tabsetting on different editors and workstations might not be the same.
- Use 2 spaces for every indentation level
- Code blocks should be aligned as such to correspond to the indentation level.
- Block declarations should be aligned so that every variable name starts at the same level.

Example:

```
    DWORD    mDword
    DWORD*   mDword_ptr
    char*    mChar_ptr
    char     mChar
```

4.3 Switch Statements

As C++ does not treat switch-cases as code blocks, and thus the break statement is a necessity, no indentation should be used. Switch statements should also use enumerations rather than integers and they should all implement a default case.

Example:

```
switch( … )
{
  case FIRST_CASE:
  <code>
  break;

  case SECOND_CASE:
  <code>
  break;

  default:
  <code>
}
```

5 General comments

- Files longer than 1000 lines should be avoided.
- 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.

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

- 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.

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

- Always initialize variables.

6 Example

The SCVS package "apptpl" contains a C++ template application that can be used as an example case and as a starting point for new developments.

___oOo___