E U R O P E A N S O U T H E R N O B S E R V A T O R Y

Organisation Européenne pour des Recherches Astronomiques dans l'Hémisphère Austral

Europäische Organisation für astronomische Forschung in der südlichen Hemisphäre

VERY LARGE TELESCOPE

VLT Data Flow System

C++ Programming Standards

Doc.No. VLT-PRO-ESO-19000-1208

Issue 1.0

Date 13/07/99

Prepared	C. Guirao, N. Kornweibel	.....	
	Name	Date	Signature
Approved	M. Peron	.....	
	Name	Date	Signature
Released	P. Quinn	.....	
	Name	Date	Signature

Change Record

Issue/Rev.	Date	Section/Page affected	Reason/Initiation/Document/Remarks
0.9	26/08/96	All	Preliminary Draft
1.0	13/07/99	Added to amendments section.	Update to final release.

TABLE OF CONTENTS

1 Introduction

1.1 Purpose

The purpose of the *C++ Programming Standards* document is to provide a common framework for the implementation phase of the VLT Data-Flow Project.

The Data Pipeline Group in his Data Flow Design Meeting from 18/04/96 took the decision to use the document *Programming in C++, Rules and Recommendations* from Ellemtel as the baseline for C++ programming in the implementation of the VLT Data-Flow System. The Ellemtel document is one of the most accepted coding standard guidelines for C++ programming.

Based on that decision a forum of discussion called *coding-std* was created with representatives of the DMD and VLT divisions and aimed to harmonize with the existing VLT software standards and to agree on amendments to the original Ellement document to better fulfill the needs of the Data-Flow project in terms of C++ Coding Standards. This document is the resume of that discussion, thus acknowledgments are therefore due to all who have contributed to this discussion.

The original Ellemtel document (copies should be available to all programmers) plus these amendments conform the *C++ Programming Standards* for the *VLT Data Flow System* and it should became a reference or a chapter of the *Quality Assurance Plan Document* for the *VLT Data Flow* System.

Included in this document is also the Software Documentation for the *VLT Data Flow System. It* includes the selected Documentation System *doc++* and templates for source and header files.

1.2 Scope

This framework is aimed, first of all, to write maintainable C++ code: consistent style within the project life cycle, easy to understand and as clear and simple as practicality allows. Second; to improve reliability: free of common bugs, resistance to the introduction of bugs and easy to test. Third, to improve portability to other architectures.

Software Documentation might not be considered part of Coding Standards, but they provide he common layout for files and format documentation that must be used in all C++ files of the project.

Design, construction, validation or verification, although very important keys in the development of a project, are beyond the scope of this document.

Coding Standards and template files are not perfect and therefore it is intended to be dynamic and should be improve over time.

1.3 Reference Documents

[1] *Programming in C++ Rules & Regulation*. Mats Henricson and Erik Nyquist (Ellemtel Telecommunication Systems Laboratories)

[2] Programming in C++. Dave Charles, dated 13/11/95 (Delcam International plc.)

[3] *C++ Documentation System- Requirements.* VLT Data Flow. (http://www.eso.org/dataflow/docs/doc_req/doc_req.html, dated 25/6/96)

[4] VLT Programming Standards VLT-PRO-ESO-10000-0228

[5] DOC++, a Documentation System for C++, rel 2.0 (http://www.ZIB-Berlin.DE/VisPar/doc++/doc++.html)

2 Amendments to the Ellemtel document

- **Rule 1**: Include files in C++ always have the file name extension ".h".
- **Rule 2**: Implementation files in C++ always have the file name extension ".C".
- **Rec. 4**: *** Removed ***
- **Exception to Rule 1**: Include files with extension ".hh" from external libraries required by the project.
- **Exception to Rule 2**: No exceptions.
- **Rec. 7**: Any include and implementation file should have a file name of the form <prefix><class name>+extension. Use uppercase and lowercase letters in the same way as in the source code. If the implementation file is split into different files the name of each new file will be the old one followed by an underscore followed by a lower case word. For example, if a file ZzzClass contains member function ZzzClass::member() which is to be implemented in a a file of its own, the file might be called ZzzClass_member.
- **Rule 10**: Never specify relative UNIX names in #include directives, except when using include files from POSIX, X11 and Motif and other third party libraries.
- **Rule 12**: The identifier of every globally visible class, enumeration type, type definition, function, constant, and variable in a class library is to begin with a prefix that is unique for the library. Prefixes are to be in lower case and with a length from two to six characters.Rule 12: The identifier of every globally visible class, enumeration type, type definition, function, constant, and variable in a class library is to begin with a prefix that is unique for the library. Prefixes are to be in lower case and with a length from two to six characters.
- **Rule 14**: The names of classes, structures are to begin with an uppercase letter. Macros, typedefs and enumerated types only use uppercase letter and if two or more words are needed use the underscore as the separator.

  Rec. 10: Use the directive **#include "filename.h"** for user-prepared include file.

  Remark: Be aware of directives #include "filename.h". You may wrongly believe that this directive force the compiler to search first in the current directory of your source file, when it searches first in the directory where the current input came from.
- **Rec. 11**: Use the directive #include <filename.h> for include files from libraries.
- **Rule. 12b**: Use the prefixes get and set for methods to access private data.

  Example:

```
class ZzzMyClass
{
  public:
    String getName() const;
    recode setName(const String& aname);
  private:
    String name;
};
```

- **Rec. 28**: Do not use tabs for code indentation and keep lines within 79 characters long. Use the code style shown in the examples of the Ellemtel document.
- **Rec. 35**: Use operator overloading sparingly and in a uniform manner. One may want to limit the set of operators which may be overloaded. In any case, if overloading is used it must be clearly noted in the documentation.
- **Rule 30a**: Do not use inheritance for parts-of-relations.

- **Rec. 37**: Avoid multiple inheritance in general or use composition instead.
- **Rec. 41**: Avoid functions with many arguments and never more than 7.
- **Rec. 46**: Minimize the number of temporary objects that are created as return values from functions or as arguments to functions. This is helped by using the prefix operations rather than the postfix ones, E.g. use ++i, rather than i++.
- **Rule 38**: Variables are to be declared with the smallest possible scope except for loops where variables can be declared just before the block of the loop.
- **Exception to Rule 43**: In addition to the Ellemtel text:
  a. Address casting away const. Use it where the implementors view of the const-ness of a variable differs from that of the user.

  Example:

```
class C {
  public:
    C():m_ember(0) {}
    ~C(){}
    void foo() const
    {
      ((C*)this)->m_ember = 15;
    }
  protected:
    int m_ember;
};
```

  An application would not consider foo() to change the (abstract) state of C, yet, internally. it does. In order for foo() to be a const function, you need to use casting.

  b. Object casting to force one of a number of ambiguous implicit conversions. (Basically, to help the compiler out.)

  c. Address-casting to a derived class. (`downcasting'). If this is done, the base class ought to know which of its derived types it is part of. An assertion that the base class is really part of the appropriate derived class should be used.

  d. .Casting in and out of a generic object (eg. void*). (This might not be necessary if templates or STL containers can be used)
- **Rec. 59**: Always assign an invalid value (e.g. 0xDEADF00D) to a pointer that points to deallocated memory. Use utility class utCheckValid for a stronger control for deallocated objects (see Appendix A).

  The theory says that it is better to have a process crash, than to spread erroneous information, so assigning an invalid address (but not 0) to a pointer of a deallocated memory it forces the process to crash (however this is not true with all systems/compilers, at least no with SunOS/g++).
- **Rec. 60**: Use exceptions to implement fault handling. An overloaded copy of operator “new” would be used to rise un exception for a “not enough memory” condition.

3 Summary of rules (with Amendments)

The following list is a summary of the C++ Coding standard Rules as appearing in the Ellemtel document, adjusted for the agreed amendments. Please note that the Summary of Recommendations provided in the Ellemtel document is also a vital element of the C++ Coding Standards.

Summary of Rules

	Rule
Rule 0	Every time a rule is broken, this must be clearly documented.
Rule 1	Include files in C++ always have the file name extension ".h"
Rule 2	Implementation files in C++ always have the file name extension ".C".
Rule 3	Inline definition files always have the file name extension ".icc".
Rule 4	Every file that contains source code must be documented with and introductory comment that provides information on the file name and its contents.
Rule 5	All files must include copyright information.
Rule 6	All comments are to be written in English.
Rule 7	Every include file must contain a mechanism that prevents multiple inclusion of the file.
Rule 8	When the following kinds of definitions are used (in implementation files or in other include files), they must ne included in separate include files: • 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. • function prototypes for functions/member functions used in inline member functions that are defined in the file.
Rule 9	Definitions of classes that are only accessed via pointers (*) or reference (&) shall not be included as include files.
Rule 10	Never specify relative UNIX names in **#include** directives, except when using include files from POSIX, X11, Motif and other third party libraries.

Summary of Rules

	Rule
Rule 11	Every implementation file is to include the relevant files that contain: • declarations of types and functions used in the functions that are implemented in the file. • declarations of variables and member functions used in the functions that are implemented in the file.
Rule 12	The identifier of every globally visible class, enumeration type, type definition, function, constant and variable is the begin with a prefix that is unique for the library. Prefixes are to be lowercase and with a length of two to six characters.
Rule 13	The names of variables, constants and functions are to begin with a lowercase letter.
Rule 14	The names of abstract data types, structures, **typedefs** and enumerated types are to begin with an uppercase letter. Macros, typedefs and enumerated types use only uppercase letters, multiple words are to be separated with underscores (‘_’).
Rule 15	In names which consist of more that one word, the words are to be written together and each word that follows the first is to begin with an uppercase letter.
Rule 16	Do not use identifiers which begin with one or two underscores (‘_’ or ‘__’).
Rule 17	A names that begins with an uppercase letter is to appear directly after its prefix.
Rule 18	A name that begins with a lowercase letter is to be separated from its prefix using an underscore (‘_’).
Rule 19	A name is to be separated from its suffix using an underscore (’_’).
Rule 20	The public, protected and private sections of a class are to be declared in that order.
Rule 21	No member functions are to be defined within the class definition.
Rule 22	Never specify public or protected member data in a class.
Rule 23	A member function that does not affect the state of an object (its instance variables) is to be declared **const**.

Summary of Rules

	Rule
Rule 24	If the behaviour of an object is dependent on the data outside the object, this data is not to be modified by const member functions.
Rule 25	A class which uses “new” to allocate instances managed by the class, must define a **copy constructor**.
Rule 26	All classes which are used as base classes and which have virtual functions, must define a virtual destructor.
Rule 27	A class which uses “new” to allocate instances managed by the class, must define an **assignment operator**.
Rule 28	An assignment operator which performs a destructive action must be protected from performing this action on the object upon which it is operating.
Rule 29	A public member function must never return a non-const reference or pointer to member data.
Rule 30	A public member function must never return a non-const reference or pointer to data outside and object, unless the object shares the data with other objects. Do not use inheritance for parts-of-relations.
Rule 31	Do not use unspecified function arguments (ellipsis notation).
Rule 32	The names of formal arguments to functions are to be specified and are to be the same both in the function declaration and in the function definition.
Rule 33	Always specify the return type of a function explicitly.
Rule 34	A public function must never return a reference or a pointer to a local variable.
Rule 35	Do not use the preprocessor directive **#define** to obtain more efficient code: use inline functions instead.
Rule 36	Constants are to be defined using **const** or **enum**: never use **#define**.
Rule 37	Avoid the use of numeric values in code: use symbolic values instead.

Summary of Rules

	Rule
Rule 38	Variables are to be declared in the smallest possible scope. Loops are an exception to this rule.
Rule 39	Each variable is to be declared in a separate declaration statement.
Rule 40	Every variable that is declared is to be given a value before use.
Rule 41	If possible, always use initialization instead of assignment.
Rule 42	Do not compare a pointer to NULL or assign NULL, use 0 instead.
Rule 43	Never use explicit type conversions (casts).
Rule 44	Do not write code which depends on functions which use implicit type conversions.
Rule 45	Never convert pointers to objects of a derived class to pointers of a virtual base class.
Rule 46	Never convert a **const** to a non-const.
Rule 47	The code following a **case** label must always be terminated by a **break** statement.
Rule 48	A **switch** statement must always contain a **default** branch which handles unexpected cases.
Rule 49	Never use **goto**.
Rule 50	Do not use **malloc**, **realloc** or **free**.
Rule 51	Always provide empty brackets (“[]”) for **delete** when deallocating arrays.

4 Software Documentation

A C++ Documentation System is defined as a tool to automate documentation generation of C++ programs.

The Documentation System selected for the Software Documentation for the VLT Data Flow System was *doc++*. This tool fulfills the requirements specified for the VLT Data Flow System [3].

Doc++ requires some conventions in the source file, like /// to mark the next identifier, and /** ...*/ to extract text. Doc++ also uses LaTex command convention to format the text and allows references to other documentation.

Only header files *.h* are required to be documented to generate a complete User Manual of the class. The implementation file *.C* keeps only documentation for maintenance and for the description of implementation details, and only using standard C++ comments //... and /*...*/.

Documentation files (HTMLE and LaTex) generated with **doc++** are kept under the directory *doc* in each subsystem . Generation of documentation file is under the make support provided by SNIFF+.

Documentation is generated per library, as a collection of classes. Three files are also used to support the generation of documentation:

- *document.doc++* References to the header files of this library
- *banner.doc++* Banner for HTML documentation
- *docxx.sty* Banner for LaTex documentation

Banner.doc++ and *docxx.sty* needs only modifications to indicate the subsystem name (e.g. VLT-DFS Subsystem Core - User Manual 1.1) and the document code (e.g. VLT-MAN-ESO-19000-0001)

Document.doc++ is the master page for the command **doc++**. It contains the list of the *include.h* files from which documentation will be generated. Optionally you may write an introduction section with a general description of your subsystem.

Detailed description for each of these files can be found in the template files available in the *~dfadmin/etc* directory.

5 Templates

5.1 Template for header files

The file *template.h* can be used as template for header files and can be found in *~dfadmin/etc* directory.

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//COPYRIGHT (c) 1996 European Southern Observatory
//LICENSE
//
//PROJECT:   VLT Data Flow System
//AUTHOR:    Carlos Guirao - ESO/DMD/DPG
//SUBSYSTEM: Core
//
//$Revision$
//$Log$
//
//-----------------------------------------------------------------------------

//C=============================================================================
//C INTRODUCTION:
//C   Class header file illustrating recommended document style for a class.
//C   Comments beginning with //C are annotations, they are not part of the
//C   recommended template!
//C
//C   Mark all identifiers (classes, methods, attributes, macros, etc..) to be
//C   included in the documentation with an extra "///" line BEFORE the
//C   definition of the identifier.
//C   The command 'docify' does exactly this! Usage: docify [infile [outfile]].
//C
//C   Optionally you use "/** <long description> */" AFTER the definition of
//C   the identifier for an extended description.
//C
//C   Only public and protected areas are included in the documentation
//C   (unless you use option -p with the command doc++ to include also
//C   private members). Standard comments may still be used anywhere, as for
//C   a class declaration it is a good practice to describe the internal data
//C   structure.
//C
//C DOCUMENTATION OF A CLASS:
//C   The documentation for a class consists of a line "/// <brief description>"
//C   BEFORE the class definition and of a LaTex formated
//C   "/** <long description> */" AFTER the class definition.
//C
//C   <brief description> should not be longer that 80 chars.
//C
//C   Sections "SYNOPSIS" and "DESCRIPTION" are required. They indicate
//C   "usage and description" of the class to the external world.
//C   DO NOT describe implementation details here!
//C
//C   All other sections are optional, remove them if they are not used.
//C   The section "SEE ALSO" may contain "\Ref{other_manual}" entries which
//C   are links to other manuals.
//C
//C DOCUMENTATION OF METHODS, ATTRIBUTES AND ARGUMENTS:
//C   Insert "///" BEFORE the identifier to include it in the documentation.
//C
//C   Optionally use "/** <long description> */" AFTER the identifier the
//C   for a complete description of methods, arguments and/or attributes.
```

```
//C
//C   The signature of a method is always included in the documentation.
//C   Methods use a LaTex formated "/** <long description> */" where only the
//C   section "Purpose" is required, other sections are optional and can be
//C   removed if they are empty. Remove itemization when not applicable.
//C
//C   Self-explaining arguments needs no comments.
//C
//C HOW TO CREATE DOCUMENTATION:
//C   You run "doc++" in a documentation directory ./doc on a file
//C   called "document.doc++". The following files can be used as templates and
//C   should be copied to the ./doc directory:
//C              ~dfadmin/etc/document.doc++
//C              ~dfadmin/etc/banner.doc++
//C              ~dfadmin/etc/docxx.sty
//C
//C   To generate HTML documentation use:
//C     % cd <subsystem>/doc
//C     % doc++ -B banner.doc++ document.doc++
//C     % netscape index.html
//C
//C   To generate LaTex documentation use:
//C     % cd <subsystem>/doc
//C     % doc++ -t -eo a4paper -o document.tex document.doc++
//C
//C SEND YOUR COMMENTS TO: Carlos Guirao ESO-DMD-DPG
//C===========================================================================


#ifndef COR_CLASS_H
#define COR_CLASS_H

// includes from this library
#include <CorBase.h>
// includes from other ESO libraries
// includes from other libraries
// system includes

/// A simple class
class CorClass : public CorBase {
  /**
  {\footnotesize $Id$ } \\
 } \\

  {\large SYNOPSIS} \\
  \begin{verbatim}
  #include <CorClass.h>
  \end{verbatim}

  {\large DESCRIPTION} \\
  This text serves as an example for the documentation of a class definition

  {\large CAUTIONS} \\
  <optional> \\

  {\large EXAMPLES} \\
  <optional> \\

  {\large SEE ALSO} \\
  <optional> \\
  \Ref{DfcClass}\\
  */
public:
```

```
  ///
  CorClass();

  ///
  ~CorClass();

  ///
  void initItsExample();
  /** {\large Purpose} \\
  A public memmber function with no arguments

  {\large Preconditions} \\
  \begin{itemize}
  \item Itemize this section if necessary. Remove itemization otherwise.
  \end{itemize}

  {\large Exceptions}\\
  \begin{itemize}
  \item Itemize this section if necessary. Remove itemization otherwise.
  \end{itemize}
  */

  ///
  int setItsExample (
    /** {\large Purpose} \\
    A public memmber function with one arguments

    {\large Preconditions} \\
    \begin{itemize}
    \item Itemize this section if necessary. Remove itemization otherwise.
    \end{itemize}

    {\large Returns}\\

    {\large Exceptions}\\
    \begin{itemize}
    \item Itemize this section if necessary. Remove itemization otherwise.
    \end{itemize}

    {\large Arguments}\\
    */
    ///
    const unsigned int&,
    /** My new value  */
    ///
    const unsigned int&
    /** My new value  */
    );
private:
  unsigned int itsExample;      // How about a comment ...
};

#endif COR_CLASS_H
```

5.2 Template for implementation files

The file t*emplate.C* can be used as template for implementation files and can be found in *~dfadmin/etc* directory.

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
//COPYRIGHT (c) 1996 European Southern Observatory
//LICENSE
//
//PROJECT:   VLT Data Flow System
//AUTHOR:    Carlos Guirao - ESO/DMD/DPG
//SUBSYSTEM: Core
//CI NUMBER: DFS-TMP-002
//
//$Name$
//$Revision$
//$Log$
//
//------------------------------------------------------------------------
static const char *rcsId = "@(#)$Id$";
static void *use_rcsId = ((void)&use_rcsId,(void *)&rcsId);

//C========================================================================
//C INTRODUCTION:
//C   Class header file illustrating recommended document style for a class.
//C   Comments beginning with //C are annotations, they are not part of the
//C   recommended template!
//C
//C DOCUMENTATION OF METHODS OF A CLASS:
//C   You do not write documentation for methods in the implementation file.
//C   It should be done in the definition of the class (.h file). The only
//C   documentation you may want to write here is implementation details.
//C   The same applies to functions which are global.
//C========================================================================

// includes from this library
#include <CorClass.h>
// includes from other ESO libraries
// includes from other libraries
// system includes

// typedef, extern, enums
// variables: consts, statics, exported variables (declared extern elsewhere)
// local forward function declarations

//========================================================================
// Purpose: Just a documentation example of a static function
// Preconditions:
// Returns:
// Exceptions:
//------------------------------------------------------------------------
static int myFunction(
  int arg1,             // something here about arg1
  int arg2)             // something here about arg2
{
}

CorClass::CorClass()
{}

CorClass::~CorClass()
{
  /* NO - OP */
}

void CorClass::initExample()
{
  itsExample = 0;
}
```

```
void CorClass::setItsExample(const unsigned int& aExample)
{
  itsExample = aExample;
}

void CorClass::printItsExample() const
{
  /* NO - OP */
}
```