Proposed Federal Aviation Administration C and C++ Coding Standards

FINAL DRAFT

Massachusetts Institute of Technology Lincoln Laboratory (MIT LL)

National Center for Atmospheric Research (NCAR)

December 2, 2009

Table of Contents

1. Introduction

This document describes a set of software coding and documentation standards for the C and C++ programming languages to be utilized in developing aviation weather systems for the Federal Aviation Administration (FAA). In the past, each of the laboratories funded by the FAA established individual software coding standards and there was no particular attempt to establish a set of common practices and guidelines. By formalizing such guidelines into an explicit set of rules, the aviation weather system software created by the laboratories should prove to be more consistent, easier to understand, extensible, and maintainable.

These rules apply to all C and C++ code with the following exceptions:

- Code produced by an automated code generator.
- Code provided by a third party.

Each rule is classified as one of the following:

- Mandatory: The rule must be followed.
- Guideline: The rule is strongly recommended, but not required.

It is recognized that no set of coding standards rules, no matter how carefully thought out, can apply to every possible programming situation. Thus, in order to allow for exceptional circumstances, this document hereby begins with the following rule:

Rule 1.1 If any rule in this document is broken, the reason why shall be clearly documented.

MANDATORY

The coding rules have been written to enable production of the best possible software. However, should the application of a rule degrade the quality of the software, then that rule may be broken and the reason should be clearly documented.

Should a project determine that a rule is not appropriate for that project, then the reason for violating the rule should be clearly stated in the project's documentation.

Reference: ESA2000 Rule 0

2. General Rules

Rule 2.1 The source code and related files shall be stored using a software version control system and all files shall contain appropriate configuration management (CM) information.

MANDATORY

In order to maintain control of the software development process, all source code files and associated files must be stored using a software version control system. Such a system should perform at least the following functions:

1. Store all historical versions of each file in a repository and provide access to them on demand.
2. Allow developers to add ("check in") new versions to the repository.
3. Keep documentation information for each version of each file. The user should be able to access a history log for each file.
4. Allow the user to "branch" the file, i.e. provide two separate modification paths for a file.
5. Allow the user to merge two modification paths back into one.
6. Provide the ability to "mark" user-chosen sets of files and versions. Such a feature might be used for milestones, releases, etc.
7. Automatically mark each file internally with user-chosen CM information.

This rule requires that function 7 above be used to ensure that each file contains useful CM information, such as the revision number, check-in date, etc.

Reference: ESA2000 Rule 1

Rule 2.2 Use tools other than compilers and linkers that can perform additional code-checking or derive more information from the code.

GUIDELINE

Compilers are limited in terms of what types of errors they can detect. For example, problems that can only be detected by examining multiple source code files simultaneously will be missed by compilers. Linkers, on the other hand, can detect some such global system issues but don't actually parse source code and thus can only have a limited understanding of the system. Developers are therefore strongly urged to use other code-checking and verification tools such as the following:

- Software that can perform additional code verification at the system level, such as the C **lint** program.
- Tools that check for memory leaks and misuse, such as Purify or Valgrind.

Reference: ESA2000 Rule 5

Rule 2.3 Use a tool that extracts information and comments from the source code and produces quality software documentation.

MANDATORY

Documentation that is current and of high quality is essential to maintaining a large software system. But it is well-known that it is difficult to keep documentation current and complete. One relatively easy way to generate lower-level software documentation is to use a tool such as Doxygen, which automatically extracts comments and various information from the source code and produces quality documentation from them. If developers are conscientious about using and maintaining comments in the source code (Rule 6.1) then useful documentation can be generated with little effort. Therefore, Doxygen or a similar tool should always be used.

Rule 2.4 Only optimize code when needed to meet documented performance requirements.

MANDATORY

Software typically has a very long lifetime and is likely to be patched, modified, and redesigned many times over that lifetime. Because of this, it is essential that the code be designed and written to be as easy to understand as possible. This way, those who maintain and modify it in the future will be able to do so without expending much of their time simply trying to understand how the code works.

However, some developers tend to add optimizations to their code that make it harder to understand even though many optimizations don't speed software up that much. In some cases, such optimizations are not necessary because the compiler will do just as good a job of optimizing itself. In other cases, optimizations are made to sections of the code that are not bottlenecks and thus the time savings are insignificant overall. The spirit of this rule is to avoid introducing overly complex optimized code that is already "good enough" from a performance standpoint. Therefore, developers should only optimize code when such optimization is necessary to meet documented performance requirements.

Reference: ESA2000 Rule 7

Rule 2.5 Instrument code for tracing and debugging purposes using a logging library. Avoid using debug macros.

MANDATORY

During the development process, programmers frequently add code for debugging purposes. Occasionally, it is useful to keep such code in the release version of the software. This should be done through the use of a logging library or class, rather than by using macros and the preprocessor. Using macros can clutter up the code and obscure the logic. For example:

```
// Using macros to maintain debugging code - not allowed
if (condition)
{
  DoSomething();
#ifdef DEBUG
    PrintSomeInfo();
#endif
}
```

The logging facility used, whether library or class, may be home-grown or acquired from a third-party source. It should be reasonably customizable via a configuration file and efficient enough that a program's execution time will not be impacted, given reasonable use of the facility.

A module should log information required for replicating a particular run of the module. Such information would typically include the location or version number of the appropriate executable, necessary environment variables, command line parameters, configuration file locations, and the like.

All log messages should be issued via calls to routines in a logging library or class. The logging levels below, listed in order of increasing severity, should be supported by the logging facility. The ability to log only levels above a specified level should be supported by the facility and configurable via a configuration file.

Debug

Debug log messages are utilized for software testing and problem investigation. The ability to suppress debug log messages during operation should be provided by the logging facility.

Informational

Informational log messages provide non-error information, such as algorithm status.

Warning

Warning log messages indicate non-fatal anomalies such that processing can continue with no degradation in service.

Error

Error log messages indicate non-fatal anomalies such that processing can continue with possible degradation in service.

Fatal

Fatal log messages indicate that an unrecoverable error has occurred such that the process must terminate.

Reference: ESA2000 Rule 9

Rule 2.6 Special debugging code should have no effect on the system state.

MANDATORY

Special code added for debugging purposes should not contain any side effects which alter the system state. It should be possible to add or remove debugging code with all outputs and behaviors remaining exactly the same.

Reference: ESA2000 Rule 10

Rule 2.7 Use appropriate software tools to help verify that the code follows the rules in this document.

GUIDELINE

Using software tools that automatically parse and review code looking for rule violations is encouraged. Such tools might bring to light code problems earlier than would have happened otherwise.

Reference: ESA2000 Rule 11

3. File and Directory Structure

Rule 3.1 Place each library and subsystem in a separate directory.

MANDATORY

Libraries and subsystems should be logically separated from each other, each exposing only its public interface to others. In order to encourage this logical separation, libraries and modules should also be physically separated. This is accomplished by placing each in its own directory. Doing this reinforces the intent to rely only on public interfaces to use other libraries and subsystems by making it more difficult for code outside of the library or module to access the implementation. Note that libraries and modules may consist of multiple interface and implementation files.

Reference: ESA2000 Rule 24

Rule 3.2 Use a template to provide the starting point for all files within each particular subsystem, library, or module.

MANDATORY

If every file adheres to some basic format then it becomes easier for other individuals familiar with that format, but not the file, in question to understand what the file is and what it does. For example, a standardized comment header that contains information about the name of the file, the last modification date, and a file description makes it easy to quickly find this information. The use of templates for new files is a way of ensuring that this format is followed from the moment a file is created.

SEE Section 19 (Sample File Format Templates) for the required templates.

Reference: ESA2000 Rules 25, 28, and 65

Rule 3.3 Source code shall be broken up into *interface* and *implementation* files.

MANDATORY

Splitting the public interface and private implementation of source code into different files allows for the clean separation of a module's interface and implementation. Having done this, another piece of source code can use the module by including just the interface file. This is desirable because that source code then only knows about the module's public interface, thereby hiding its private implementation. The interface file shall have an ".h" or ".hh" suffix. The implementation file shall have either a ".c", ".cc", ".C", or ".cpp" suffix. The particular suffixes chosen should be consistent across a project.

To illustrate this rule, consider a circle class that has an X coordinate, Y coordinate, radius, a constructor that sets the member variables, and a member function to draw

the circle. The interface is in a file *Circle.h* which contains the following class declaration:

```
class Circle
{
  public:
    Circle(double x, double y, double radius) :
      x(x), y(y), radius(radius) {}
    void draw();

  protected:

  private:
    double x;
    double y;
    double radius;
};
```

The implementation is in *Circle.C*, which contains the definition of the function to draw the circle:

```
#include "Circle.h"

void Circle::draw()
{
  // Code to draw the circle
}
```

Now, other modules can use the Circle class by including just the interface file. There is no need to manually declare it again or bring in all the code to draw a circle into the module:

```
#include "Circle.h"

int main()
{
  Circle bigCircle(0, 0, 100);
  bigCircle.draw();
}
```

Reference: ESA2000 Rule 26

Rule 3.4 Each implementation file shall include its own interface file.

MANDATORY

By including the interface file in the appropriate implementation file, the compiler can verify that the two are consistent with each other. Doing this also gives the implementation file access to information in the interface file it may need, such as a class declaration.

Reference: ESA2000 Rule 27

Rule 3.5 Interface files shall only include the necessary interface files.

MANDATORY

Each interface file shall include all the interface files required by items that appear in the file. No other files shall be included. Including all necessary interface files ensures that when the interface file in question is itself included by another file it will compile and link successfully. Unnecessary interface files should not be included in order to maximize the degree to which components of the program are decoupled.

Additionally, forward declarations shall be used instead of including an interface file where possible. For example, if a class *ClassA* in *InterfaceA* contains a pointer to a class *ClassB* in *InterfaceB*, and *InterfaceA* in no other way refers to *InterfaceB*, then use a forward declaration of *ClassB* in *InterfaceA* and do not include *InterfaceB* at all as in this example:

```
// No need to #include InterfaceB

class ClassB;     // Forward declaration

class ClassA
{
  public:

  protected:

  private:
    ClassB* classBPtr;
}
```

Reference: ESA2000 Rule 29

Rule 3.6 Preprocessor "guard" directives shall be used around the contents of interface files to prevent problems that would arise from an interface file being included multiple times.

MANDATORY

It is possible for a single interface file to be included into another file multiple times. For example, if *ImplementationA* includes *InterfaceA* and *InterfaceB*, and *InterfaceA* also includes *InterfaceB*, then *ImplementationA* will have two copies of *InterfaceB* after the preprocessor substitutes the contents of the files in. This can result in errors and warnings due to duplicate declarations, definitions, and preprocessor constructs, as well as increased file sizes and longer compile times.

In order to avoid this problem, the contents of every interface file should be wrapped in a "guard" directive. These preprocessor constructs specify that the

enclosed contents will only be included if the preprocessor macro in question has not already been defined. The use of a guard directive is illustrated in this example:

```
#ifndef INTERFACE_A
#define INTERFACE_A

// Contents of interface file

#endif
```

The preprocessor variable chosen shall be unique for every interface. As such the name should be based on the name of the interface file or class it contains.

Reference: ESA2000 Rule 30

Rule 3.7 The interface file shall not contain any function definitions.

MANDATORY

If a function were to be defined in an interface file then every file which includes that interface file would have its own definition of the function. This can result in unexpected behavior if the function contains a static variable as each version of the function would then have its own static variable. Additionally, duplicate function definitions result in an unnecessary degree of redundancy, larger files, and increased compile time.

EXCEPTION Inline function definitions may be present in the interface file.

EXCEPTION Function templates or member functions of class templates may be defined in the interface file.

Reference: ESA2000 Rule 31

4. Compiling and Building Software

Rule 4.1 Compile and build libraries and programs by using an automated tool such as the Unix *make* command.

MANDATORY

The process of collecting source code files and libraries together and compiling and linking them into libraries and programs is typically too complex to be done manually. Instead, an automated "build" tool such as the Unix **make** command should be used. This will ensure that the software is built in a complete and consistent manner by the various developers on the team and over time.

Reference: ESA2000 Rule 2

Rule 4.2 Do not use non-standard language features.

MANDATORY

All compilers support extensions, enhancements, and other features that are not officially part of the programming language. Such features are not guaranteed to be portable across different hardware, operating systems, compilers, or compiler versions and should not be used.

Reference: ESA2000 Rule 4

Rule 4.3 Programs shall produce no compiler warnings during the build process.

MANDATORY

Compilers typically produce many warnings—messages that are not strictly errors but indicate that the source code contains possible problems or undesirable features. Warning messages should be taken seriously, as the program may work properly for the moment (or at least appear to) but may misbehave at any time in the future due to code flagged by them. Moreover, even if some warnings are verified as being benign, leaving them unaddressed leads to more cluttered build logs and makes it more likely that developers won't notice important messages. Therefore, code that produces warning messages shall be modified so that the message no longer appears.

Reference: ESA2000 Rule 6

5. The C and C++ Preprocessor

Rule 5.1 System header files shall be included using angle brackets; user header files shall be included using quotation marks.

MANDATORY

Examples of proper use of the **#include** statement:

```
#include <system.h>  // System header file
#include "user.h"    // User header file
```

This makes it easy to determine which files are system files and which are user files, as it may not be apparent from the name. Also, on some systems, the style of brackets used affects how the file system is searched for include files.

Reference: ESA2000 Rule 108

Rule 5.2 Include C++ headers rather than C-style header files.

MANDATORY, C++ ONLY

ISO C++ defines the standard implementation of its libraries and the official headers that must be included to make use of them. Programmers should use these headers rather than custom versions or the deprecated C-style header files. For example:

```
#include <iostream.h>      // Incorrect
#include <stdio.h>         // Incorrect
#include <iostream>        // Correct
#include <cstdio>          // Correct
```

NOTE This rule only addresses C++ headers. System header files still will take the form:

```
#include <socket.h>        // Correct
```

Reference: PRL2008 Rule 17.1

Rule 5.3 Absolute path names shall not be used when including a header file.

MANDATORY

If the absolute path name is used in an **#include** statement, all the source files using the referenced header file will need to be updated if the header file is moved. In addition, different systems have different directory structures, so **#include** statements may need to be updated when porting the code to a new system. Instead of using absolute paths, the programmer should only reference the file name itself. The build system can be used to specify the directories to search (see Rule 4.1).

The problems cited above may occur even if a partial path name is used, so it is suggested that they also be avoided when possible. However, it is recognized that

this is not always possible. For example, many Unix header files require a partial path, for example:

```
#include <sys/time.h>
```

Reference: ESA2000 Rule 109

Rule 5.4 Use preprocessor commands and macros to conditionally compile non-portable code.

MANDATORY

This allows the same file to be compiled on different systems, making it easier to configure builds and easily produce executables intended for different systems. Here is an example:

```
#ifdef UNIX
  // UNIX-specific code
#elif WINDOWS
  // Windows-specific code
#else
#error ERROR: This code only works on UNIX and Windows.
#endif
```

Reference: ESA2000 Rule 110

Rule 5.5 Wrap preprocessor macros and macro parameters in parentheses if they contain parameters or expressions.

MANDATORY

The preprocessor runs before the compiler. Preprocessor macros are replaced through simple text substitutions which occur before the code is actually compiled. Because of this, macros can cause unexpected side effects if not defined properly. For example:

```
#define DOUBLE(x) x+x
DOUBLE(3) * 2     // Evaluates to 3+3 * 2 = 9 rather than 12
```

This problem can be avoided if the expression is enclosed in parentheses:

```
#define DOUBLE(x) (x+x)
DOUBLE(3) * 2     // Evaluates to (3+3) * 2 = 12
```

However another problem may arise related to handling of the parameters:

```
#define SQUARE(x) (x * x)
SQUARE(2+3)       // Evaluates to (2+3 * 2+3) = 2+6+3 = 11
                  // rather than 25
```

For this reason, each instance of a parameter should also be enclosed in parentheses:

```
#define SQUARE(x) ((x)*(x))
SQUARE(2+3)       // Evaluates to ((2+3)*(2+3)) = 25
```

Reference: ESA2000 Rule 111

Rule 5.6 Do not use macro parameters containing expressions that may have side effects.

MANDATORY

Macros cause the C preprocessor to do simple text substitutions, so unexpected effects can occur when certain expressions are used as parameters. For example, the order of evaluation of parameters does not apply to macros because they are not functions. In addition, expressions with side effects can be problematic as in the following:

```
#define SQUARE(x) ((x)*(x))
SQUARE(y++);      // Evaluates to (y++)*(y++), probably not
                  // desired
```

Reference: ESA2000 Rule 112

Rule 5.7 The preprocessor shall not be used to hide or change basic C or C++ constructs.

MANDATORY

Programmers who are familiar with another programming language may wish to use macros to map C or C++ into this language, essentially redefining the language. Such an attempt might look something like the following:

```
#define IF   "if"
#define THEN "{"
#define ELIF "} else if"
#define ELSE "} else {"
#define FI   "}"

IF (2 == x)
THEN
  foo1(x);
ELIF (3 == x)
THEN
  foo2(x);
ELSE
  foo3(x);
FI
```

This is forbidden because it makes the code difficult to maintain, especially if a future programmer is not familiar with this language. Also, many support tools (e.g., syntax-aware editors) will not be able to work with the macros.

Reference: ESA2000 Rule 113

6. Comments and Inline Documentation

Rule 6.1 Comments in the source code shall be relevant and be kept up to date to be in sync with the source code.

MANDATORY

Out-of-date comments can mislead developers trying understand a program. Thus, comments shall be maintained so that if the source code changes, all relevant comments will be updated or removed as necessary.

Reference: ESA2000 Rule 8

Rule 6.2 Write all comments in English.

MANDATORY

All comments shall be written in English whenever possible. If a non-English word must be used in a comment, a sufficient English definition shall be given in the comment.

Reference: ESA2000 Rule 12

Rule 6.3 Use the "/*" and "*/" comment delimiters in C programs.

MANDATORY, C ONLY

For files written in the C language, "/*" and "*/" comment delimiters shall be used. For single line comments, use "/*" as a starting delimiter and "*/" as an ending delimiter on that line. For multi-line comments, the following useful convention is suggested: "/*" should be used as a starting delimiter on its own separate line preceding the comment content. For the comment content between the "/*" and "*/" on multiple lines, "*" should be used as a marker at the beginning of each line. Finally, "*/" should be used as an ending delimiter on its own separate line. Though not strictly necessary, the "*" on the intervening lines are useful as they will show up when file searches are done, e.g. by the Unix **grep** command.

Examples:

```
/* This is a single-line comment in a C program. */
/*
 * This is a multi-line
 * comment
 * in a C program.
 */
```

Reference: ESA2000 Rule 13

Rule 6.4 Place Doxygen comment blocks describing classes, functions, and interfaces in the interface files.

MANDATORY

For all source code written in a language that uses interface files, such as C and C++, any Doxygen-style comment blocks describing classes, functions, or interfaces shall be placed in those header files. The intention is to provide all necessary documentation for interfacing with the code in the header file, while comments in the source files are reserved to provide more lower-level and specific help in reading and understanding the code.

Examples of function and class descriptions to be placed in header files can be found in Section 19 (Sample File Format Templates).

Rule 6.5 Use the "//" comment delimiter and/or the Doxygen style "/**" and "*/" comment delimiters when using C++.

MANDATORY, C++ ONLY

For files written in the C++ language, "//" comment delimiters shall be used at the beginning of each line of a comment block.

Doxygen comments in the JavaDoc style may also be used. In this style, "/**" shall be used as a starting delimiter on its own separate line preceding the comment content. For the comment content in between the "/**" and "*/" on multiple lines, "*" shall be used as a marker at the beginning of each line. Finally, "*/" shall be used as an ending delimiter on its own separate line.

Examples:

```
// This is a single-line comment in a C++ program.

// This is a multi-line
// comment in a
// C++ program.

temp = 32;        // This is an inline comment

/**
 * This is a Doxygen JavaDoc style comment
 * in a C++ program.
 */

int temp;         /**< This is an inline Doxygen comment */
```

Reference: ESA2000 Rule 14

Rule 6.6 Each function shall have a sufficiently descriptive comment.

MANDATORY

For every function in an interface file, some sort of comment shall be provided to sufficiently describe how to interface with and utilize the function. Any function parameters and function return values should be described. Even simple short functions or inline functions shall include a comment block. Using Doxygen-style comments for function descriptions is required. See Rule 19.5 for an example of the format that should be used.

To prevent duplication, functions in implementation files shall not have comments repeating any of the above-mentioned information. However, comments describing *how* the function works should be included as needed.

Reference: ESA2000 Rule 60

Rule 6.7 Comments shall not be in an elaborate layout.

GUIDELINE

Comments should not be written in any sort of elaborate layout. This commonly includes extra characters to provide additional separators and delimiters in the comment. Complex layouts may look appealing but require more time and effort to maintain. This may cause comments to be of a lower quality or even become out of sync with the code, due to the effort needed to get the elaborate formatting correct. All comment blocks should instead follow the rules and examples in this document.

Reference: ESA2000 Rule 15

Rule 6.8 Comments shall contribute to and not detract from the understanding of the code.

MANDATORY

In order for comments to be useful, their content needs to be meaningful in a way that makes the source code easier to understand. Superfluous or excessive comments that do not contribute to the understanding of the code should be avoided. The intent is not to reduce the number of comments, but rather to eliminate unnecessary comments that provide obvious or irrelevant information. These unhelpful comments simply take up space and may distract a reader.

Here are some examples of comments that detract from understanding the code:

```
int a;          // stores the value of a
a++;            // increments the value of a
foo();          // calls function foo
```

Reference: ESA2000 Rule 16

Rule 6.9 Comments shall not be used to disable code.

MANDATORY

A widely used programming practice is to use comments to disable unused code for development purposes. While this practice is acceptable during development, comments shall not be used in this manner in completed code. Code that is not used should be removed completely.

Example of unacceptable use:

```
// Comments are being used to disable this code,
// which is unacceptable for completed code.
// foo()
// {
//   bar();
// }
```

Reference: ESA2000 Rule 17

7. Code Layout

Rule 7.1 The length of a line shall be reasonably limited in order to improve readability.

MANDATORY

In order to support readability on common editor and terminal sizes, a single consistent and reasonable maximum line length shall be used for all source code files. This may aid in comparing files side by side, as well as avoiding line wrapping when printing. A maximum line length of 80 characters is suggested.

Maximum line lengths may differ between projects, but only a single maximum line length should be used within a project.

EXCEPTION A line in a comment that may be commonly copied and pasted, such as an example command or a URL.

Reference: ESA2000 Rule 18

Rule 7.2 Enclose every nested block of code, including one-line blocks, within braces.

MANDATORY

Example:

```
if ( condition )
{
  DoSomething();
}
```

Reference: ESA2000 Rule 19

Rule 7.3 The standard brace style shall be used.

MANDATORY

Braces will have their own line and will line up with each other under the statement's keyword:

```
if ( condition )
{
  DoSomething();
  DoSomethingElse();
}
else
{
  DoOneThing();
}
```

Reference: ESA2000 Rule 20

Rule 7.4 The standard indentation style shall be used.

MANDATORY

Indent two spaces within blocks:

```
while ( condition )
{
  DoSomething();
}
```

Reference: ESA2000 Rule 21

Rule 7.5 Use these styles for control statements.

MANDATORY

if statements:

```
if ( condition )
{
  DoSomething();
}
else
{
  DoSomethingElse();
}
```

switch statements:

```
switch ( expression )
{
  case value1:
    DoSomething1();
    break;

  case value2:
    DoSomething2();
    // Fall-through comment

  case value3:
  case value4:
    DoSomething3();
    break;

  default:
    DoDefaultThing();
}
```

for statements:

```
for ( int i = 0; i < N; i++ )
{
  DoSomething();
}
```

while statements:

```
while ( condition )
{
  DoSomething();
}
```

do while statements:

```
do
{
  DoSomething();
} while ( condition );
```

try catch constructs (C++ only):

```
try
{
  Class::DoSomething();
}
catch ( Class::Exception exception )
{
  Complain();
}
```

NOTE The spaces before and after each "(" and ")" are optional.

Reference: ESA2000 Sections 6.2-6.6

Rule 7.6 A *break* statement or a comment explaining its absence shall be included in each case within a *switch* statement.

MANDATORY

Generally speaking, each **case** statement in a **switch** statement should end with a **break** statement because the typical **switch** statement consists of a set of mutually exclusive actions.

However, when one **case** statement performs some processing and then "falls through" to the next **case** statement, the absence of a **break** must be explained in a comment. This improves code readability and maintainability as the reader will not wonder if the **break** was omitted by mistake. For an example, see case *value2* in Rule 7.5.

EXCEPTION If a **case** statement does not perform any processing, but simply falls through to the next case, it does not need to be commented. A fall-through such as this is immediately apparent. See case *value3* and case *value4* in Rule 7.5.

Reference: ESA2000 Rule 22

Rule 7.7 Always use a *default* label at the end of a *switch* statement.

MANDATORY

Even if the programmer does not believe that the **default** label can be reached, including one will help to catch errors early in development, and may help catch data corruption which may cause unexpected values in the **switch** expression.

Reference: ESA2000 Rule 23

Rule 7.8 Use these styles for declarations.

MANDATORY

Structures:

```
struct StructType
{
  type1 member1;
  type2 member2;
  ...
};
```

Classes (C++ only): See Rule 19.4.

Class templates (C++ only):

```
template < typename T1, typename T2 >
class Class
{
  // members
};

Class < type1, type2 > class1;
```

Unions:

```
union UnionType
{
  type1 member1;
  type2 member2;
  ...
};
```

Enumerated types:

```
enum EnumType { identifier1, identifier2, ... };

enum EnumType
{
  identifier1,
  identifier2,
  ...
};
```

NOTE The spaces before and after each "<", "{", ">", and "}" are optional.

8. Naming Conventions

For the purposes of Rule 8.5, Rule 8.6, Rule 8.7, Rule 8.8, and Rule 8.10, the term "word" is defined to mean "any string of letters and numbers". It is expected that such a word will have a clear meaning to the reader of the program as required by Rule 8.2.

Rule 8.1 The underscore character "_" shall not be used as a prefix to names.

MANDATORY

The rules for names that begin with underscores are restrictive. C and C++ variously reserve names that begin with one or two underscores for the compiler, operating system, libraries, and global namespace. It is in some cases technically legal to make a name with a leading underscore. However, due to the complexity of these rules it is safer to never begin a name with an underscore. Moreover, even if it were safe to use leading underscores, it would be inadvisable because it can be difficult to tell at a glance how many consecutive underscores there are in a name.

Reference: ESA2000 Rule 32

Rule 8.2 Names shall be descriptive, accurate, and consistent.

MANDATORY

Names should not be random but rather refer to what the variable holds, function does, class represents, etc. For example, a variable holding a running total as a data structure is stepped through should have a name along the lines of *total* or *sum*, not *abc*. The name should be representative throughout its entire lifetime, e.g. *total* should not later be reused as a counter for a loop. Additionally, names for similar constructs should be consistent with each other to prevent confusion:

```
// Incorrect: names are inconsistent
int nodeCount;              // Number of nodes
int edgeNum;                // Number of edges
int isolatedNodes;          // Number of nodes with no edges

// Correct
int nodeCount;              // Number of nodes
int edgeCount;              // Number of edges
int isolatedNodesCount;     // Number of nodes with no edges
```

Reference: ESA2000 Rule 33

Rule 8.3 Abbreviations used within names shall be consistent and unambiguous.

MANDATORY

It is a common practice to abbreviate parts of names so as to keep the name's length reasonable. This has the potential to create confusion and errors when an abbreviation which is obvious to one individual is unclear or means something else to another:

```
// Incorrect: ambiguous abbreviation
int opts;            // Old points, Options, or Operations?

// Better: still confusing
int oldpts;          // Old Points
int optns;           // Options
int oprtns;          // Operations

// Correct
int oldPts;          // Old Points
int options;         // Options
int operations;      // Operations
```

Additionally, abbreviations shall be consistent. Using different abbreviations to mean the same thing throughout a program is likely to result in errors as it is difficult to remember which form of the word to use where:

```
// Incorrect: inconsistent abbreviations
int cmpnntName;      // Name of the component
int componentType;   // Type of the component
int cmpntSize;       // Size of the component
int cmptPtr;         // Ptr to the component

// Correct
int cmpntName;       // Name of the component
int cmpntType;       // Type of the component
int cmpntSize;       // Size of the component
int cmpntPtr;        // Ptr to the component
```

Reference: ESA2000 Rule 34

Rule 8.4 Names which are very similar to each other shall not be used.

MANDATORY

Names that are almost identical to one another or mean nearly the same thing are a common source of errors and confusion. Of particular concern is the difference between "1" (the number one) and "l" (lower-case L), as well as "0" (the number zero), and "O" (upper-case O). Variable names that differ by only replacing one of these characters with the other shall not be used:

```
// Incorrect names
int length1;     // First length
```

```
int lengthl;      // Length of "L"
int value0;       // Value of zero
int valueO;       // Value of "O"
```

Likewise, avoid differentiating between variables by only replacing words in one with others that mean the same thing (for example: *lengthX* vs. *sizeX*).

Reference: ESA2000 Rule 35

Rule 8.5 Variable names shall be comprised of one or more words concatenated together, where the first letter of every word except the first is capitalized.

MANDATORY

To ensure uniformity throughout all source code, it is important that the names of variables adhere to some standard format. The convention selected here is to capitalize the first letter of each word or abbreviation in the name except the first and not use any underscores or other characters between the words. Doing this makes the name readable and distinguishes variables from classes and user-defined types, each of which begins with a capital letter.

Here are some examples:

```
int I;            // Incorrect
int i;            // Correct
int Count;        // Incorrect
int count;        // Correct
int HeadCount;    // Incorrect
int head_count;   // Incorrect
int headCount;    // Correct
int numLines;     // Correct
```

Reference: ESA2000 Rule 40

Rule 8.6 Function names shall be comprised of one or more words where the first letter of each word is capitalized.

MANDATORY

EXCEPTION The first word may or may not be capitalized. Consistency in this matter shall be on the project level and the chosen methodology shall be identified and documented.

Like variable names, function names should follow a standard format. The convention chosen is identical to that used for variable names, except that the first word may also be capitalized.

Here are some examples:

```
void copy();          // Depends on project-specific rules
void Copy();          // Depends on project-specific rules
void copy_item();// Incorrect
```

```
void Copy_Item();// Incorrect
void copyItem(); // Depends on project-specific rules
void CopyItem(); // Depends on project-specific rules
```

Reference: ESA2000 Rule 41

Rule 8.7 Type names shall be comprised of one or more words where the first letter of each word is capitalized plus an appropriate prefix or suffix.

MANDATORY

Like variable and function names, user-defined type names should follow a standard format. In addition, it is useful to distinguish user-defined type names from variable names. System libraries have traditionally done this by ending type names with "_t". Therefore it is logical and consistent to end user-defined type names with "_t" as well. Sometimes, however, it is argued that user-defined type names should be better differentiated from system-defined ones, so type names may also be specified by using "T" as a prefix or "_type" as a suffix. Other than this prefix or suffix, the convention chosen is identical to that used for variable names except that the first letter of the name is also capitalized. This is done to further distinguish user-defined types from built-in types.

Here are some examples:

```
typedef char byte;                  // Incorrect
typedef char Byte_t;                // Correct
typedef char Byte_type;             // Correct
typedef int* ptr_to_int;            // Incorrect
typedef int* T_PtrToInt;            // Correct
```

Reference: ESA2000 Rules 42 and 43

Rule 8.8 Class names shall be comprised of one or more words where the first letter of each word is capitalized.

MANDATORY

Like variable and function names, user-defined class names should follow a standard format. The convention chosen is identical to that used for variable names except that the first letter of the name is also capitalized. This is done to distinguish user-defined class names from the names of variables, built-in types, and STL classes.

Here are some examples:

```
class tree;                         // Incorrect
class Tree;                         // Correct
class binarySearchTree;             // Incorrect
class Binary_search_tree;           // Incorrect
class BinarySearchTree;             // Correct
```

Reference: ESA2000 Rule 42

Rule 8.9 Consistent prefixes shall be used on named constants within an enumeration.

MANDATORY

It is possible for multiple enumerations with similar characteristics to be used within a program. For example, there may be more than one enumeration representing the status of various parts of the program. To avoid name clashes and problems when converting between the enumeration and an integer, a prefix corresponding to the name of the enumerated type should be attached to each named constant.

For example:

```
// Incorrect
typedef enum
{
  Unknown,
  On,
  Off
} PowerStatus_t;

// Correct
typedef enum
{
  PowerUnknown,
  PowerOn,
  PowerOff
} PowerStatus_t;
```

Reference: ESA2000 Rule 45

Rule 8.10 Preprocessor names shall consist of one or more words where all letters are upper-case and underscores are used to separate the words.

MANDATORY

Preprocessor variables are different from actual C and C++ constructs. Occurrences of a preprocessor name (i.e. a macro) are replaced by the preprocessor with the appropriate value or expression before compilation begins. In order to highlight this difference, preprocessor names shall consist of only upper-case characters and digits and underscores shall be used to separate words so that the name is readable.

Here are some examples:

```
#define FilterParamsH      // Incorrect: Lower-case letters
#define FILTERPARAMSH      // Incorrect: Difficult to read
#define Filter_Params_H    // Incorrect: Lower-case letters
#define FILTER_PARAMS_H    // Correct
```

Reference: ESA2000 Rule 39

9. Declarations

Rule 9.1 Each declaration shall be placed on a separate line with a comment describing the entity being declared.

GUIDELINE

In order to increase readability and maintainability, each declaration should be on its own line and include a comment that describes the entities created. This comment does not need to be in line; it is sufficient for the comment to be nearby. If multiple variables are being declared, each declaration should only instantiate one variable at a time (see Rule 9.10).

Reference: ESA2000 Rule 46

Rule 9.2 Entities shall be declared to be visible in the most limited scope and with the shortest lifetime possible.

GUIDELINE

Scope refers to the enclosing context in which an entity may be defined. Although scoping behavior may differ from language to language, generally ensuring that entities are declared in the narrowest scope within reason helps avoid possible conflicts among different sections of code. Limiting access to the entity to only the necessary client code may also help prevent undefined behavior through unintended use. In the case of variable declarations, code understandability may also be improved by placing the declarations closer to where the variables are actually used.

Determining the appropriate scope for an entity may also take into account factors such as performance and maintainability. Declaring an entity at the most specific possible local scope may not always be the best solution.

Lifetime refers to length of time that, if applicable, an entity will actually exist in memory. The lifetime of such entities should be minimized to improve performance, particularly in memory allocation. A common example would be to create an array with memory allocated in the heap and then free the memory immediately when the array is no longer needed.

Reference: ESA2000 Rule 47

Rule 9.3 Declarations of all global entities shall be placed in an interface file.

MANDATORY

A global entity is an entity that is visible from every scope and can be accessed and modified from anywhere in any implementation file. Each declaration of a global entity shall be placed in a relevant interface file so that any implementation file wishing to access the global entity must include the appropriate interface file. However, global entities should be avoided whenever possible (Rule 18.1).

Reference: ESA2000 Rule 48

Rule 9.4 All declarations of external variables and functions shall be placed in an interface file.

MANDATORY

An external variable or function, signified by the **extern** keyword, is a variable or function with external linkage that can be declared or defined in any file. Each declaration of an external variable or function shall be placed in an appropriate interface file so that such declarations can be shared easily and consistently among the files that need access to such entities.

Reference: ESA2000 Rule 49

Rule 9.5 No static variables or functions shall be declared in interface files.

GUIDELINE

Static variables are variables whose lifetime spans the entire runtime of the program, but which may have their scope limited. Such static entities must appear only in implementation files. If a static variable appeared in an interface file, it would be defined in *each* file that includes that interface file, i.e. there would be multiple definitions of the entity. This leads to code that is difficult to maintain and debug. In addition, in many cases, this is likely not what the programmer actually wanted anyway.

Static functions are functions that are not exported to the linker, i.e. are not known outside the given file. As is the case with static variables, if a static function appeared in an interface file, it would be defined in *each* file that includes that interface file. This would be inefficient and could lead to unexpected behavior in certain circumstances.

Reference: ESA2000 Rule 50

Rule 9.6 Declarations shall appear in the order: (1) constants and macros, (2) type definitions, (3) structures, unions, and classes, (4) variables, and (5) functions.

GUIDELINE

Declarations shall be made in a specific order for consistency and to improve maintainability. This order will be: (1) constants and macros, (2) type definitions, (3) structures, unions, and classes, (4) variables, and (5) functions. This ordering is practical as each category of entities frequently builds upon the previous category. Finally, logically-related declarations in the same category should be organized together for better readability.

Reference: ESA2000 Rule 51

Rule 9.7 Symbolic constants shall be used in the code. “Magic” numbers and strings are expressly forbidden.

MANDATORY

Symbolic constants are identifiers that are used to represent literal values that can be numbers, characters, or strings. This improves the program’s readability and maintainability, because it helps define the meaning and usage of a literal value, and provides a single point of change for that literal value. Such values whose meaning and usage are not immediately obvious are "magic" numbers and strings and shall never be written directly into the code, except to define the aforementioned symbolic constants.

Reference: ESA2000 Rule 52

Rule 9.8 Symbolic constants shall be defined in C using a preprocessor macro, the *const* keyword, or an enumerated type.

MANDATORY, C ONLY

When using the C language, symbolic constants shall be defined either using a preprocessor macro, the **const** keyword, or an enumerated type. Selecting which method to use depends on how the literal value of the symbolic constant will be used. The preprocessor macro simply replaces the symbolic constant identifier with the exact characters that were defined during compilation. Using the **const** keyword actually creates an entry in the compiler's symbol table during compilation and also prevents the value from being modified. The preprocessor macro is convenient for simple single constant values, but if the literal value is a more complicated expression, then **const** may be superior. Using **const** can have other benefits; for example, such names will be known in a debugger while macros will not. An enumerated type is often useful when a set of sequential and/or related symbolic constants or a bit mask of symbolic constants is needed.

Here are some examples of constants:

```
#define SYM_CONSTANT 1024
const int SYM_CONSTANT = 1024;

enum SYM_CONSTANTS
{
  CONSTANT1,
  CONSTANT2,
  CONSTANT3
};

enum SYM_CONSTANTS
{
  CONSTANT1 = 1,
  CONSTANT2 = 2,
  CONSTANT3 = 4
};
```

Reference: ESA2000 Rule 53

Rule 9.9 Symbolic constants shall be defined in C++ using the *const* keyword, an inline function that returns the constant, or an enumerated type.

MANDATORY, C++ ONLY

When using the C++ language, symbolic constants shall be defined either using the **const** keyword, an inline function that returns the constant, or an enumerated type. Selecting which method to use depends on how the literal value of the symbolic constant will be used. Using the **const** keyword will prevent the literal value from being modified. The inline function call will be used in lieu of the symbolic

constant, but the function call, in turn, will be replaced by the actual body of the inline function. Such an inline function returning a constant shall be used when it is necessary to ensure correct initialization of static constants. The enumerated type is often useful when a set of sequential or related symbolic constants or a bit mask of symbolic constants is needed.

Here are some examples of constants:

```
const int SYM_CONSTANT = 1024;

inline int getSymConstant()
{
  return 1024;
}

enum SYM_CONSTANTS
{
  CONSTANT1,
  CONSTANT2,
  CONSTANT3
};

enum SYM_CONSTANTS
{
  CONSTANT1 = 1,
  CONSTANT2 = 2,
  CONSTANT3 = 4
};
```

Reference: ESA2000 Rule 54

Rule 9.10 Declarations shall only contain one variable or constant each.

MANDATORY

Each variable shall have its own declaration in order to increase readability regarding the variable's type and initial value (if any). In the following example, *a* is declared as type **char[]** and *b* is declared as type **char** in the same declaration. This may be confusing and is thus not allowed.

```
char a[], b;
```

EXCEPTION In the cases of loop counters and logically-grouped variables, more than one variable may appear on a line.

Reference: ESA2000 Rule 55

10. Standard Template Library Containers

Rule 10.1 Provide efficient copying for objects in containers.

GUIDELINE, C++ ONLY

Class *copy constructors* are called whenever class-type member objects are added to a container. Class copy assignment operations may also be invoked when an object is moved inside of some container types (such as **vector**). For these reasons, copy operations should be implemented in an efficient manner.

For example:

```
class Foo
{
  public:
    Foo();
    Foo( Foo const& rhs );            // copy constructor
    Foo& operator=( Foo& rhs );       // copy assignment
};

void foo( const Foo& obj, const Foo& obj2 )
{
  std::vector< Foo > vec;

  // This will cause the copy constructor to be called:
  vec.push_back( obj );

  // This will result in the copy assignment operator
  // being called for each object stored after the
  // insert iterator:
  vec.insert( vec.begin(), obj2 );
}
```

Reference: PRL2008 Rule 17.3

Rule 10.2 Use containers of pointers or smart pointers to reduce the expense of copying.

GUIDELINE, C++ ONLY

If copying is expensive, (e.g., if a container class design would involve including large data members), consider using pointers or smart pointers instead. Containers of pointers are efficient for operations such as insertion and sorting, since pointers have built-in operators for copying values and are small. Note that if containers of pointers are used, then the programmer is responsible for managing the life of the objects. A reference-counting smart pointer class can be used for this.

Reference: PRL2008 Rule 17.4

Rule 10.3 Derived class objects shall not be inserted into a container designed to hold base class objects.

MANDATORY, C++ ONLY

If derived class objects are inserted into a container designed to hold base objects, the derived objects will be *sliced* (information not in the base class will be lost) and therefore they will not hold the complete intended object. This problem can be avoided by storing pointers to base class objects.

Reference: PRL2008 Rule 17.5

Rule 10.4 Use *empty()* to test if a container has no elements instead of checking *size()* against zero.

MANDATORY, C++ ONLY

Although they are functionally equivalent, **empty()** shall be used instead of **size()**, since **size()** involves calculating the number of elements and this can be expensive for some containers.

For example:

```
std::vector<int> v;

// Correct use of empty() to check for empty container
if ( v.empty() == false )
{
  doProcessing();
}

// Incorrect.  Don't use size().
if ( v.size() != 0 )
{
  doProcessing();
}
```

Reference: PRL2008 Rule 17.6

Rule 10.5 Don't derive classes using an STL container as the public base class.

MANDATORY, C++ ONLY

STL containers do not have virtual destructors, so a class that is derived from one may have undefined behavior. This can happen, for example, if the derived class is destructed through a base class reference.

Example:

```
// Inappropriate derivation from std::map
class DerivedMap : public std::map {};

void someFunc()
{
```

```
  // Allocate derived object
  DerivedMap* derivedMapPtr = new DerivedMap;

  // Access of derived object through base class
  std::map* baseMapPtr = derivedMapPtr;

  // Deletion results in undefined behavior!
  delete baseMapPtr;
}
```

Reference: PRL2008 Rule 17.7

Rule 10.6 Don't put *auto_ptr* objects inside STL containers.

MANDATORY, C++ ONLY

Elements in STL containers must provide copy functions that result in equivalence between the source and destination. However, **auto_ptr** objects manage the pointers associated with them, so when an **auto_ptr** object is copied, ownership is "transferred" (no two **auto_ptr** objects should own the same element) and the source loses its value.

Here is an example of this problem:

```
class SomeClass{};

void someFunc( std::vector< std::auto_ptr<SomeClass> >& vec )
{
  std::auto_ptr<someClass> obj1 = vec[0];

  // obj1 is a null pointer after assignment to obj2
  std::auto_ptr<someClass> obj2 = obj1;
}
```

Reference: PRL2008 Rule 17.8

Rule 10.7 (a) Use STL vectors in place of dynamically-allocated one-dimensional arrays. (b) Use STL strings in place of dynamically-allocated *char* arrays used for text processing.

MANDATORY, C++ ONLY

The **vector** and **string** containers automatically manage their storage, provide many commonly-needed operations, and operate with the various STL generic algorithms. These containers significantly reduce the programming burden associated with managing dynamically-allocated arrays and provide efficient and reliable code for doing so.

Here are some examples:

```
// Preferred use of vector:
std::vector<int> myVec;
```

```
for ( int i = 0; i < 10; i++ )
{
  myVec.push_back(i);      // Safe
}

// Incorrect.  Don't use dynamic arrays.
int* myArr = new int[5];
for ( int i = 0; i < 10; i++ )
{
  myArr[i] = i;  // Will overflow when i > 4
}

// Example use of string for dynamic string building
std::string goodString;
goodString.append( "Good" ).append( " string." );

// Incorrect.  Don't use dynamic char arrays.
char* badString = new char[8];
strcpy( badString, "Bad string." );   // Memory overflow
```

Reference: PRL2008 Rule 17.9

Rule 10.8 Pre-allocate container storage where possible to save unnecessary reallocations later.

MANDATORY, C++ ONLY

STL containers automatically increase their storage capacity as needed when additional elements are inserted. However, these capacity increases can be costly as they involve the allocation of memory and the potential movement of previously inserted elements. Where possible, reserve the required capacity for the container in advance to reduce the need for subsequent incremental capacity increases.

For example:

```
static const int maxElements = 100;
std::vector<int> vec;

vec.reserve( maxElements );
```

Reference: PRL2008 Rule 17.10

Rule 10.9 When passing a vector to a function that expects an array, use the address of the first element of the vector.

MANDATORY, C++ ONLY

The elements of a non-empty STL vector are stored contiguously and can be used as C-style arrays if they contain C-compatible data types. To do this, use the address of the first element in the vector as a pointer to an array of elements. The syntax is

```
&v[0]
```

where *v* is a vector of some C-compatible type. Do not attempt to use other methods of treating the vector as an array since these are implementation-specific and are not guaranteed to be portable.

Here is an example:

```
extern "C" void cFunc( int i[] );

void cppFunc( vector<int>& vec )
{
  if ( vec.empty() == false )          // Must be non-empty
  {
    // Pass the vector to the C function to be used as an
    // array.
    cFunc( &vec[0] );
  }
}
```

Reference: PRL2008 Rule 17.11

Rule 10.10 Use the *c_str()* function of the C++ string class to produce a *const char** if one is needed for legacy functions.

MANDATORY, C++ ONLY

The **string::c_str()** function returns a valid, null-terminated C-style string that can be used where a **const char*** is needed. Don't use alternative methods of producing a C-style character representation as they may be implementation-specific and are not guaranteed to be portable.

For example:

```
void printString( const char* theString )
{
  cout << theString << endl;
}

int main( int argc, char** argv )
{
  std:string hello( "Hello" );
```

```
    printString( hello.c_str() );
}
```

NOTE If a **char*** is needed instead, then **const_cast** must be used also.

Reference: PRL2008 Rule 17.12

Rule 10.11 Do not use a vector of *bool* values.

MANDATORY, C++ ONLY

The STL **vector<bool>** class is not simply a **vector** class that can hold **bool** elements but rather a specialization of the **vector** class template that's implemented in quite a different manner from other vector classes. Because of this, it does not conform to all requirements of a container and certain typical operations that work on most containers do not work on **vector<bool>** objects. In particular, one cannot take the address of an element and thus the following will not work correctly:

```
vector<bool> vb;
func(&v[0]);      // Won't produce an array of bools
```

In lieu of **vector<bool>**, use **dequeue<bool>**.

Reference: PRL2008 Rule 17.13

Rule 10.12 Once a key is inserted into a set or multiset, that key should never be modified.

MANDATORY, C++ ONLY

The elements of sets and multisets are sorted based on their keys. An element is inserted into the correct position when it is inserted into the data structure. Changing the key of an element does not cause it to resort. Consequently, were a key to be changed, it might result in the set or multiset no longer being properly sorted, likely causing the container to function improperly or even cause a crash.

Reference: PRL2008 Rule 17.15

Rule 10.13 Use *push_back()* to add elements to the end of a container.

GUIDELINE, C++ ONLY

There are a number of ways of adding elements to a sequential STL container, but the **push_back()** member function is the most efficient of these, always operating in constant time. In addition, using **push_back()** is clearer than using functions such as **insert()**.

Reference: S&A2005 Rule 80

Rule 10.14 Use *remove()* and *erase()* together to shrink capacity and truly erase elements.

GUIDELINE, C++ ONLY

The STL **remove()** algorithm does not actually erase elements from a container. It just reorders the elements and modifies the pointers. To really remove an element (i.e., destroy it) and reduce the size of the container, the call to **remove()** needs to be followed by a call to **erase()**:

```
std::vector<int> c;

//…

int value = 3;
// Remove all values equal to 3 from the vector
c.erase( remove(c.begin(),c.end(), value), c.end() );
```

If the STL container has a member version of **remove()**, this should be preferred over the generic algorithm version. The member version will destroy the elements, so the call to **erase()** would not be necessary.

Reference: S&A2005 Rule 82

11. Functions

Rule 11.1 All forward declarations of functions shall use the ANSI style. The traditional "Kernighan & Ritchie style" [K&R1998] is forbidden.

MANDATORY

The old style of forward declaration of functions was particularly error-prone because an integer return type was assumed if no return type was specified. Also, no consistency checking was done between the function call and actual declaration of the number and type of parameters. Forward declaration of functions must use the ANSI style, which requires the return type and all parameters to be specified.

Example:

```
// A function
float foo(int param1)
{
  //…
}

// Bad prototype
foo();

// Bad prototype
float foo();

// Good prototype
float foo(int param1);
```

Reference: ESA2000 Rule 58

Rule 11.2 A full function prototype for each function with global scope shall be declared in the interface file(s).

MANDATORY

ANSI C and C++ style prototypes provide consistency checking between the prototype and function call or actual definition of the function. Putting prototypes in the interface file provides organization, showing programmers what functions are available and how to call them without them needing to search through source files. It is also simpler and more maintainable to include the interface file instead of putting the function prototype in each source file the programmer wants to call the function from.

Reference: ESA2000 Rule 59

Rule 11.3 An explicit return type shall be declared for each function. A return type of *void* shall be declared for any function which returns no value.

MANDATORY, C ONLY

Using an explicit return type shows the programmer how the function works. It helps avoid errors by allowing type checking of the return value. It also prohibits the compiler from assuming an integer return type, which it would do if no return type was specified.

Reference: ESA2000 Rule 61

Rule 11.4 A *void* argument shall be declared for any function which takes no parameters.

MANDATORY, C ONLY

If a function takes no parameters, it should be made explicit by declaring it to have a "void" argument. This should also be reflected in the function prototype, thereby avoiding confusion with the old "K&R" style of function prototype [K&R1998], which did not specify the function's parameters.

Example:

```
// Incorrect declaration
int foo()
{
  //…
  return 1;
}

// Correct declaration
int foo(void)
{
  //…
  return 1;
}
```

Reference: ESA2000 Rule 62

Rule 11.5 Any function parameter which is not modified by the function shall be *const* qualified in the function declaration.

MANDATORY

This helps the compiler catch errors. Declaring a parameter as **const** if it should not be modified allows the compiler to issue a warning if it is modified by accident.

EXCEPTION It is not necessary to **const** qualify parameters which are passed by value because the function cannot modify the value of the parameter outside of the function's local scope.

EXCEPTION In cases where the contract of the function allows a parameter to be modified, that parameter shall not be const-qualified. In other words, when determining whether to qualify a parameter with **const**, the contract of the function takes priority over the actual operation of the function.

Reference: ESA2000 Rule 63

Rule 11.6 Function parameters shall be validated.

GUIDELINE

By validating parameters, the function can exit if a problem is detected, which avoids unexpected errors later in the program and assists in debugging. This is usually done using the **assert()** facility, which aborts the program with some diagnostic information if it is passed a zero value. This is enabled during development and turned off before release by defining a certain preprocessor variable before compilation. Several examples are below, but parameter validation may include other checks not mentioned here.

Example, checking the size of inputs:

```
class ThreadData
{
  //…
};

void startThreads( vector<ThreadData> inputs,
                   int numOfThreads )
{
  // Make sure we have input for each thread
  assert( inputs.size() == numOfThreads );
  //…
}
```

Example, checking the type of inputs:

```
class ThreadData
{
  //…
};
```

```
class serverThreadData : public ThreadData
{
  //…
};

class clientThreadData : public ThreadData
{
  //…
};

enum threadType { serverThread, clientThread };

void startThread(ThreadData* input,
                 threadType type )
{
  // Make sure we have correct type of input
  if( type == serverThread )
  {
    assert( typeid(*input) == typeid(serverThreadData) );
  }
  else
  {
    assert( typeid(*input) == typeid(clientThreadData) );
  }

  //…
}
```

Reference: ESA2000 Rules 64 and 105

Rule 11.7 References or pointers to local automatic variables or (for C++ only) static local variables shall not be returned from functions.

MANDATORY

Local automatic variables are allocated when a function is entered and returned to the system when the function exits. Therefore it is not guaranteed that pointers or references to these variables will be valid outside of the function. The memory may have been overwritten or a segmentation fault may occur when the memory is accessed.

Static local variables are not returned to the system on exit from the function. However, it is bad practice to return references or pointers to them. This creates confusing code which is difficult to maintain.

Programmers should not try to avoid this rule. See Rule 11.8 for more information.

Reference: ESA2000 Rule 66

Rule 11.8 A reference or pointer to memory allocated by a function may not be returned from the function.

GUIDELINE

To try to avoid the limitation of returning pointers or references to local variables, a programmer may wish to dynamically allocate memory within a function and return a pointer or reference to this memory. This should be avoided.

When a function allocates memory, it should always return the memory to the system. If it returns a pointer or reference to this memory, it becomes the responsibility of the caller to clean up the memory. This can easily introduce memory leaks, especially when a programmer other than the function's author is calling the function. The caller may not even be aware that the function allocates memory.

References make this even more confusing because references can be used as if they were objects that were returned by value. This makes it very easy to forget that the memory was dynamically allocated and must be returned to the system.

Reference: ESA2000 Rule 83

Rule 11.9 The *inline* keyword shall be associated with a function only if one or more of the following holds: (1) The function requires optimization as determined by a performance analysis of the system, (2) The function is a class accessor ("get" function) or mutator ("set" function), (3) The function is very small.

MANDATORY

Inline functions are not compiled like regular functions. The code of an inline function is placed where it is called from, which typically replicates the code. If the function is long, this can greatly increase the size of the compiled program. However, inline functions are appropriate in the following special circumstances:

1. Inline functions are faster than regular functions because the program does not have to jump to the function. If a function is meant to be called repeatedly, then execution time can be shortened if it is inline.
2. Accessors and mutators are simple, often one line functions that simply return or set a value. They are used often, so are usually inline functions.
3. If a function is very small, the time to jump to and back from the function may be a large part of the function's execution time. Making a small function inline will cause a noticeable decrease in the function's execution time. However, for larger functions this jump time will be a very small part of the execution time. Thus, making a large function inline will not lead to a significant performance increase and will greatly increase the size of the compiled program.

Reference: ESA2000 Rule 84

12. Classes

Rule 12.1 A class definition shall have exactly one (possibly empty) member declaration group for each of the access specifier keywords (*public*, *protected*, *private*). These groups shall be declared in the following order: (1) *public*, (2) *protected*, (3) *private*.

MANDATORY, C++ ONLY

By using exactly one of each access specifier, in the same order each time, the layouts of class declarations becomes standardized, which makes it easier for an individual to parse the contents of a class at a glance. Were there permitted to be any number of the specifiers in any order, it would be difficult, for example, to quickly locate all of the private members of the class. Including a specifier even when it is not strictly required makes it immediately clear that no members fall under that access level. In addition this removes the temptation to simply append the specifier to the end of the class, thereby breaking the standardizing ordering, when it is needed at a later time. Here is an example illustrating this layout:

```
class Class
{
  public:
    // Public members

  protected:
    // Protected members

  private:
    // Private members
};
```

Reference: ESA2000 Rule 67

Rule 12.2 Classes shall be used rather than *structs*.

GUIDELINE, C++ ONLY

When collecting items into aggregates, classes are preferable to structures as classes are a core element of the object-oriented paradigm. Technically, one can implement a class using either the **class** or **struct** keyword, however the former better conveys the intention.

Reference: ESA2000 Rule 56

Rule 12.3 Any class member variable that is not *const* qualified shall not be public. Accessors and mutators shall be used to access and change private members.

MANDATORY, C++ ONLY

Sometimes, access to a class member variable from another class is desired. It is possible to provide this by making that variable public. However, doing so exposes the implementation of the class and might give users of the class more access than is needed. For example, in some cases it may be necessary to perform some check on the value being assigned to the member variable.

In order to support necessary member access and create a standardized format for accessing member variables, accessors ("get" functions) and mutators ("set" functions) shall always be used instead of making the variable public. Not only does this keep the implementation hidden, but if additional work needs to be done when the variable is accessed, it can easily be added to that accessor or mutator function. For example, a class may store a value in meters but the accessor could be designed to return it in miles. Here is an example of a simple class declaration containing an accessor and mutator for a variable:

```
class Class
{
  public:
    int x() const {return x;}                    // Accessor
    void x( const int value ) {x = value;}       // Mutator

  protected:

  private:
    int x;
};
```

EXCEPTION Accessors and mutators need not be used for classes that are behaviorless aggregates (i.e. C-style structures). Assigning and returning data to and from their members is all such aggregates do. It would be redundant and add complexity to give their members accessors and mutators.

Reference: ESA2000 Rule 68

Rule 12.4 All classes shall contain an explicitly declared constructor.

MANDATORY, C++ ONLY

A default constructor is supplied by the complier if no other constructor is declared. This constructor takes no arguments, has no initializer, and has nothing in its body. Member values are consequently initialized to zero. Although this may suffice in many cases, some member variables might not accept a value of zero and, for others, zero may not be the most appropriate value to use. Therefore, in order to

avoid such issues as well as create a standard format for the declaration of classes, a constructor that initializes the member variables as necessary shall always be explicitly declared:

```
class Class
{
  public:
    Class();      // Non-default constructor

  protected:

  private:
    int x;
    int y;
};

Class::Class() : x(0), y(0)           // Set x to 0, y to 0
{
}
```

Reference: ESA2000 Rule 69

Rule 12.5 A class shall contain an explicitly declared destructor.

MANDATORY, C++ ONLY

A default destructor is supplied by the compiler if no other destructor is supplied. While the default destructor may destroy the automatic member variables and objects of the class, it will not free any dynamically-allocated memory. Therefore, to avoid memory leaks and to create a standard format for the declaration of classes, a destructor that returns memory to the system as necessary shall always be declared. For example:

```
class Foo
{
  public:
    Foo(int size);
    ~Foo();       // Non-default destructor

  protected:

  private:
    int* data;
};

Foo:Foo(int size)
{
  data = new int[size];
}
```

```
Foo:~Foo()
{
  delete[] data;
}
```

Reference: ESA2000 Rule 70

Rule 12.6 A class' destructor must be virtual unless the class does not contain any other virtual member functions.

MANDATORY, C++ ONLY

Virtual functions are usually used when derived classes are going to be accessed through a pointer to their base class. Usually the destructor will be accessed the same way. However, if the base class destructor is not virtual, it may not be called correctly:

```
const int SIZE = 50;

class Base
{
  public:
    Base();
    virtual void print() = 0;

    ~Base();

  // …
};

class Derived : public Base
{
  public:
    Derived()
    {
      data = new char[SIZE];
    }

    print()
    {
      cout << data.c_str() << endl;
    }

    ~Derived()
    {
      delete[] data;
    }

  //…
```

```
  private:
    char* data;
};

Base* obj1 = new Derived();
Derived* obj2 = new Derived();

delete obj1;      // Incorrect: ~Derived is not called
delete obj2;      // Correct: ~Derived is called
```

In the above example, because the *Base* destructor is not virtual, the *Derived* destructor is not called when *obj1* is freed. If the *Base* destructor were virtual, then the derived class' destructors would be invoked.

Reference: ESA2000 Rule 71

Rule 12.7 A class shall declare a copy constructor if the default copy constructor is not suitable.

MANDATORY, C++ ONLY

The default bitwise copy constructor may cause problems when dynamic memory allocation is used. Consider the following example:

```
class DataContainer
{
  public:
    DataContainer(int size)
    {
      data = new char[size];
    }

    ~DataContainer()
    {
      delete[] data;
    }

  // Uses default copy constructor

  //…

  private:
    char* data;
};

bool doFunctionOnData(DataContainer dataLocal)
{
  //…
}

DataContainer dataGlobal;
```

```
doFunctionOnData(dataGlobal);
```

The *DataContainer* object is passed by value to the function, so the default copy constructor is called. The values of all the member variables are copied, so the data pointers of the objects *dataLocal* and *dataGlobal* point to the same memory. When the function exits, the destructor for *dataLocal* is called, which frees this memory. Now the data pointer of *dataGlobal* is a "dangling" pointer: it points to memory which is no longer valid. By declaring a copy constructor, this dynamic memory can be copied to the new object to avoid this problem as in the following example:

```
DataContainer::DataContainer(const DataContainer& src)
{
  data = new char[size];

  strncopy(data, src.data, size);
}
```

Reference: ESA2000 Rule 72

Rule 12.8 When initializing class members in a constructor initialization list, the order of initialization shall match the order in which the members are declared in the class definition.

MANDATORY, C++ ONLY

Standardizing the layout of declarations and initializations improves code readability and maintainability. Moreover, data members are initialized in order of their declaration not in the order of the initialization list. Listing them in a different order in the initialization list misleadingly implies that they will be initialized in that order. For example:

```
class Foo
{
  //…
  int x;
  int y;
  int z;
};

// Incorrect, order of initializations does
// not match order of declarations
Foo::Foo(int size) : y(size), z(size), x(size)

// Correct
Foo::Foo(int size) : x(size), y(size), z(size)
{
  //…
}
```

Reference: ESA2000 Rule 73

Rule 12.9 A constructor with an initialization list shall correctly construct the object regardless of the order of evaluation of individual statements in the initialization list.

MANDATORY, C++ ONLY

Data members must not be initialized with other data members that are in the same initialization list. This can cause confusion:

```
class Foo
{
  //…
  int x;
  int y;
  int z;
};

Foo::Foo(int size) : y(size), z(y), x(y)
{
  //…
}
```

This does not produce the correct output because data members are initialized in order of their declaration, not in the order of the initialization list. So in this example, *x* will be initialized to 0, while *y* and *z* will be initialized to size.

SEE Rule 12.8 which discusses proper initialization order.

Reference: ESA2000 Rule 74

Rule 12.10 If possible, objects shall be initialized during construction using copy constructors.

MANDATORY, C++ ONLY

Initializing an object when it is constructed is more efficient than constructing then initializing it afterwards. If it is not initialized during construction, an object with default values will be created. Later, when it is initialized via an assignment operator, these default values will all be overwritten, which wastes time, especially when the classes are large. For example:

```
// oldObject created and initialized earlier
// Preferred
largeClass newObject1(oldObject);
largeClass newObject2 = oldObject;

// Avoid if possible
largeClass newObject3;
newObject3 = oldObject;
```

Reference: ESA2000 Rule 75

Rule 12.11 A class shall declare an assignment operator when the default assignment operator is not suitable.

MANDATORY, C++ ONLY

The default assignment operator can cause problems, especially when classes have member pointers. Consider the following:

```
class Container
{
  public:
    Container() { }
    Container(char* theData)
    {
      data = new[strlen(theData) + 1];
    }

    ~Container()
    {
      delete[] data;
    }

  private:
    char* data;
};

Container c1("the data");
int foo()
{
  Container c2;  // c2.data = NULL;
  c2 = c1;
  // Now c2.data and c1.data point to the same data

  //…
} // End of scope, c2 is destroyed
```

When *c2* is destroyed at the end of this function, memory is cleaned up. This leaves *c1* with a dangling pointer (one that points to invalid memory). This can be avoided by using an explicit assignment operator, for example:

```
Container& Container::operator=(const Container& src)
{
  delete[] data; // Free old memory, if any

  data = new char[strlen(src.data) + 1];
  strcpy(data, src.data);

  return(*this);
}
```

Reference: ESA2000 Rule 76

Rule 12.12 The assignment operator(s) shall not assign an object to itself.

MANDATORY, C++ ONLY

When copying memory in the assignment operator (such as in the example in Rule 12.11), problems can occur if the source and destination are the same. For example, memory contents may be lost if they are freed before they are copied. Moreover, even if no errors will occur, assigning an object to itself, member by member, is clearly wasteful. Thus, the assignment operator should detect this situation:

```
Container& Container::operator=(const Container& src)
{
  if( this == &src )
  {
    return *this;
  }
  //…
}
```

Strictly speaking, what actually happens here is that the assignment operator *allows* an object to be assigned to itself but does not actually perform any action in that case.

Reference: ESA2000 Rule 77

Rule 12.13 Base class member data shall be assigned in the assignment operator(s) of a derived class.

MANDATORY, C++ ONLY

When an assignment operator is called, the whole object should be copied, which includes the base class data. Base class data should not be copied on a member by member basis by the derived class because, if a change is made in the base class, the assignment operator of the derived class must be updated. Also, the base class may have private members which the derived class does not have access to. Instead, the assignment operator of the base class should be used by the derived class. If required, it can be declared a protected member instead of a public member so that only derived classes have access to it. For example:

```
class DerivedClass : public BaseClass
{
  //…

  DerivedClass& DerivedClass::operator=(const DerivedClass&
                                        src);
};

DerivedClass& DerivedClass::operator=(const DerivedClass& src)
{
  // Make sure we are not assigning the object to itself
  if( &src == this )
```

```
  {
    return *this;
  }

  // Assign the BaseClass data members
  BaseClass::operator=(src);

  // Copy DerivedClass member data here

  return *this;
}
```

Reference: ESA2000 Rule 78

Rule 12.14 The assignment operator shall return a reference to the object on which assignment is being performed.

MANDATORY, C++ ONLY

Returning a reference allows assignments to be chained in one statement:

```
dataObj1 = dataObj2 = baseDataObj;
```

Reference: ESA2000 Rule 79

Rule 12.15 Unary operators and assignment operators shall be overloaded by defining class member functions. Other binary operators shall be overloaded by defining non-member or *friend* functions.

MANDATORY, C++ ONLY

Ideally, every overloaded operator function should be a member of the relevant class. However, certain properties of C++ make this impossible to do properly in the case of binary operators other than assignment operators.

First, we consider unary operators. When overloading operators using class member functions, the first operand is always the object the function is a member of, which maps well to unary operators:

```
class Point
{
  public:
    Point(double x, double y) : x(x), y(y) {}
    void operator++() {++x; ++y;}

  protected:

  private:
    double x;
    double y;
};
```

Next, we consider assignment operators. The first operand (**this**) and the operand from the argument list (if any), are handled differently when overloading operators using class member functions; **this** can be modified but the operand from the argument list cannot. Because of this asymmetric behavior, assignment operators can also be successfully overloaded by defining class member functions:

```
class Point
{
  public:
    Point(double x, double y) : x(x), y(y) {}
    Point operator+=(const Point& rhs)
      {x += rhs.x; y += rhs.y;}

  protected:

  private:
    double x;
    double y;
};
```

Finally, we consider all other binary operators. Such operators must be overloaded by defining non-member or *friend* functions because the operation is symmetrical and neither operand should be modified by the operation:

```
class Point
{
  public:
    Point(double x, double y) : x(x), y(y) {}
    friend Point operator+(const Point& lhs,
                           const Point& rhs);
  protected:

  private:
    double x;
    double y;
};

Point operator+(const Point& lhs, const Point& rhs)
{
  return Point(lhs.x + rhs.x, lhs.y + rhs.y);
}
```

Reference: ESA2000 Rule 80

Rule 12.16 Use the *const* modifier when declaring any member function that does not modify the state of the object.

MANDATORY, C++ ONLY

Declaring a member function **const** signifies to the compiler that the function will not modify the object it is a member of. If the function does modify the object, the compiler will produce an error message. Examples of such functions might include simple accessors and those that calculate something based upon the object's data and/or the given parameters. The following class contains a number of **const** functions:

```
class Line
{
  public:
    // Explicit constructor
    Line(int x1, int y1, int x2, int y2) :
         x1(x1), y1(y1), x2(x2), y2(y2) {}

    // Simple accessors
    double getx1() const {return x1;}
    double gety1() const {return y1;}
    double getx2() const {return x2;}
    double gety2() const {return y2;}

    // Calculate the distance
    double calcDistance() const
      {return sqrt( pow( x2 - x1, 2 ) + pow( y2 - y1, 2 ) );}

  protected:

  private:
    double x1;
    double y1;
    double x2;
    double y2;
};
```

EXCEPTION A member function that returns a non-**const** pointer or non-**const** reference to member data shall not be declared **const**. But note that Rule 12.17 forbids such functions.

EXCEPTION In cases where the contract of the function allows the object to be modified, the function shall not be declared **const**, even if the function currently does not modify the object. In other words, when determining whether to declare a member function **const**, the contract of the function takes priority over the actual operation of the function.

Reference: ESA2000 Rule 81

Rule 12.17 No public member function shall return a reference or pointer to a private class members unless the reference or pointer is *const* .

GUIDELINE, C++ ONLY

As stated in Rule 12.3, member variables shall be declared **private** so that access to them can be controlled. If a public member function were to return a non-**const** pointer or non-**const** reference to a member variable, then whoever receives that pointer or reference would be able to directly modify the variable without going through the class interface. This violates the principle that the class implementation be invisible and inaccessible. For example, it would allow the circumvention of any range-checking or other work done by the member functions that are supposed to be used to access and modify the variable in question, and could ultimately result in placing the object in an invalid state.

Reference: ESA2000 Rule 82

13. Exceptions

Rule 13.1 Each exception that can potentially be thrown by a called function shall be explicitly caught by the calling function in a *try-catch* construct and be either handled locally or re-thrown to some other handler.

MANDATORY, C++ ONLY

Exceptions are good ways to handle errors and make programs more robust, but there is no point to using exceptions if they are not caught and handled properly. Consider a function which can throw two exceptions, *Ex1* and *Ex2*:

```
void foo() throw (Ex1, Ex2)
{
  //…
  throw Ex1();
  //…
  throw Ex2();
  //…
}

// Two incorrect ways of calling foo

// Exceptions are not handled
void goo()
{
  //…
  foo();
  //…
}

// Only some exceptions are handled
void goo()
{
  try
  {
    foo();
  }
  catch (Ex1)
  {
    cerr << "caught Ex1" << endl;
  }
}

// Two correct ways of calling foo

// All exceptions are handled locally
void goo()
{
```

```
  try
  {
    foo();
  }
  catch (Ex1)
  {
    cerr << "caught Ex1" << endl;
  }
  catch (Ex2)
  {
    cerr << "caught Ex2" << endl;
  }
}

// goo doesn't know how to handle Ex2
void goo() throw (Ex2)
{
  try
  {
    foo();
  }
  catch (Ex1)
  {
    cout << "caught Ex1" << endl;
  }
}
```

Reference: ESA2000 Rule 88

Rule 13.2 The function *set_unexpected()* shall be used to specify which user-defined function is to be called if an unforeseen exception is caught. This must be done in every program which uses exceptions.

MANDATORY, C++ ONLY

While all possible exceptions that a function can throw should be listed in its exception specification, they may not all be listed for various reasons. For example, the function is from old code that didn't follow these standards, the function is from third-party software, etc. The function **set_unexpected()** should be used to specify what to do in case an unexpected exception is thrown. The function takes as its argument a pointer to a function with no parameters that returns **void**.

Example:

```
class Ex1
{
  //…
};

class Ex2
```

```
{
  //…
};

void handleUnexpected()
{
  cerr << "Caught unexpected exception." << endl;
};

void badFunction() throw (Ex1)
{
  throw Ex2();
}

int main()
{
  set_unexpected(handleUnexpected);

  try
  {
    badFunction();
  }
  catch (Ex1)
  {
    cout << "Caught Ex1" << endl;
  }

  return 0;
}
```

In the above example, no provision is made to catch *Ex2*. Thus, when *badFunction()* throws *Ex2*, *handleUnexpected()* will be called because it was passed to the **set_unexpected()** function.

Reference: ESA2000 Rule 89

Rule 13.3 The function *set_terminate()* shall be used to specify which user-defined function is to be called if a problem arises while handling an exception.

MANDATORY, C++ ONLY

The most common problem that arises with exception handling is that a handler for the exception cannot be found. When such a problem arises during exception handling, the system will call the function that was set using **set_terminate()**. This function needs to be set even if an "unexpected" function was set through **set_unexpected()**. The default behavior is for the unexpected function (called when an unexpected exception is caught) to call the "terminate" function, which in turn calls **abort()** to end the program. While this paradigm is useful, by replacing the

defaults with user-defined functions, diagnostic information can be printed and/or saved to assist in debugging.

The **set_terminate()** function takes the same argument as the **set_unexpected()** function, a pointer to a function with no arguments that returns **void**.

Example:

```
void myTerminate()
{
  cerr << "Terminate handler called" << endl;
  abort();        // Forces the program to exit
}

int main()
{
  set_terminate(myTerminate);

  // Unhandled exception: calls terminate handler
  throw 0;
  exit 0;
}
```

Reference: ESA2000 Rule 90

14. Expressions

Rule 14.1 A value of arithmetic or enumeration type shall not be converted to a value of type *bool*. A Boolean may be obtained from a value of arithmetic or enumeration type by performing a comparison on the value.

MANDATORY

Relying on the implicit comparison of values against zero, although appealing in its briefness, can lead to confusion and strange errors. Consider the following code:

```
if (performTest())
  // Test passed, do something
```

If *performTest()* simply returned a Boolean, with **true** indicating success and **false** failure, this would be fine. However, if *performTest()* instead returned some other data type (e.g. an integer), it would be unwise to rely upon the implicit comparison to zero. The problem is that interior of the **if** statement will be reached as long as *performTest()* returns any value except zero. Consequently, if the function were to return negative numbers to indicate various errors, those error codes would be interpreted as passing the test. Other functions may use zero to indicate what might be considered 'success' (i.e. **strcmp()** returns zero if the strings are equivalent). Additionally, relying on the implicit conversion of enumeration types to Booleans can be dangerous as the enumeration may be renumbered at some point. Therefore, to avoid these kinds of errors and sharpen the meaning of the code, this rule requires that Booleans be obtained from other types by performing explicit comparisons:

```
enum Status {SUCCESS, FAILURE, ERROR};

Status performTest()
{
  // If something goes wrong
    return ERROR;
  // If test passes
    return SUCCESS;
  // Else
    Return FAILURE;
}

if (performTest() == SUCCESS)
  // Do something
```

Reference: ESA2000 Rule 92

Rule 14.2 Every expression should be written to ensure that the order of evaluation of the expression is well-defined and unambiguous.

MANDATORY

With the exception of certain operators, used as *sequence points* and outlined below, the order in which expressions within a statement are evaluated is undefined. If such expressions have side effects that may affect other parts of the expression, undefined behavior may result. The sequence point operators are:

1. The comma operator (“,”) enforces that the expression to its left is evaluated before the expression to its right. This is not the comma used to separate function arguments; that comma is not a sequence point and its arguments could be evaluated in any order.
2. The *logical and* operator (“&&”) will evaluate the expression on its left first, and then only evaluate the expression on the right if the left expression was true.
3. The *logical or* operator (“||”) will evaluate the expression on its left first, and then only evaluate the expression on its right if the left expression was false.
4. The conditional operator (as in “a ? b : c”) will evaluate the leftmost expression first. If the leftmost expression is true, then only the middle expression is evaluated afterwards. If the leftmost expression is false, then only the rightmost expression is evaluated.

It is important to keep these rules in mind in order to avoid writing ambiguous code. The following lines of code have undefined results:

```
functionCall(x, y, x+=y);

array1[i++] = array2[i];
```

Reference: ESA2000 Rule 93

Rule 14.3 Use parentheses to make the order of evaluation of operators in an expression clear.

GUIDELINE

In order to better communicate his or her intentions, the programmer should use parentheses to indicate the order of operations in expressions in potentially confusing cases—even when the expression is unambiguous to the compiler. For example:

```
a = x + y * z;       // Unclear
a = x + ( y * z );   // Equivalent to first expression but
                     // clearer
a = ( x + y ) * z;   // Parentheses required in this case
```

Reference: ESA2000 Rule 95

Rule 14.4 Use parentheses around bitwise operators.

GUIDELINE

It is easy to introduce errors using bitwise operators because they behave as arithmetic operators in some sense, but have a much lower precedence. Always using parentheses around a bitwise operator and its operands helps to avoid these errors. Consider the following code to pick out a particular bit from a word and compare it to zero:

```
if (stateWord & bit4Mask == 0)
```

The equality operator binds more tightly than the *bitwise and*, so this code is actually testing *bit4Mask* (which is necessarily non-zero) against zero for equality then taking the *bitwise and* of that result and *stateWord* as if we had written this:

```
 if (stateWord & (bit4Mask == 0))
```

Adding parentheses around the bitwise operation causes the code to behave as expected:

```
if ((word & bit) == 0)
```

Reference: ESA2000 Rule 96

Rule 14.5 White space shall not be used around the "." and "->" operators or between a unary operator and its operand.

MANDATORY

The proper use of white space can greatly improve the readability of code. In order to better communicate to the reader the order of operations, space can be left between loosely-coupled operators and their operands while no space should be left between more tightly-coupled operators and their operands. The "." and "->" operators are among the highest precedence operators in C and C++, having the same ranking as parentheses and square brackets, and should therefore have no white space around them. The unary plus and minus operators rank just below them and thus should not have white space between them and their operands.

```
// Incorrect
point1 . x = 10;
point1ptr -> x = NULL;
point1.x = - 10;

// Correct
point1.x = 10;
point1ptr->x = NULL;
point1.x = -10;
```

Reference: ESA2000 Rule 97

Rule 14.6 Traditional C-style casts shall not be used.

MANDATORY, C++ ONLY

The C-style casts override type information about the variables or pointers being cast. This may cause portability problems, e.g. a particular cast may not be valid on a system, but the compiler will perform the cast anyway. The C++ style casts should be used instead because they make clear the intention of the cast. Also, the C++ style casts can easily be searched for in source code files, unlike C-style casts. There are four C++ casts:

static_cast is equivalent to the old C-style cast, but it makes it more obvious what is intended:

```
static_cast<type>(expression)
```

dynamic_cast performs runtime type-checking and is used when casting within a class hierarchy (from a base class to a derived class):

```
ptr = dynamic_cast<type>(expression)
```

const_cast removes (or adds) the **const** or **volatile** attribute of the expression:

```
const_cast<type>(expression)
```

reinterpret_cast reinterprets the expression as the new type. But the resulting value will have the same bit pattern as the original expression:

```
reinterpret_cast<type>(expression)
```

Reference: ESA2000 Rule 91

Rule 14.7 Operators shall be surrounded by white space where appropriate to improve readability.

MANDATORY

As mentioned in Rule 14.5, white space can be used to enhance the readability of code, especially in regard to the operators within an expression. Separating loosely-coupled portions of an expression with white space makes the code easier to comprehend:

```
// Incorrect
answer = w * x+y * z+( a - b );
answer = w * x + y * z + ( a - b );
answer = w*x+y*z+(a-b);

// Correct
answer = (w * x) + (y * z) + (a - b);
answer = w*x + y*z + (a-b);
```

EXCEPTION The tightly-coupled operators referenced in Rule 14.5.

Reference: ESA2000 Rule 98

15. General Usage of the Standard Template Library

Rule 15.1 STL containers and algorithms shall be used, unless performance requires a custom design.

MANDATORY, C++ ONLY

STL containers and algorithms are part of the C++ language standard. They have been well tested, can be implemented much faster than custom designs, and are generally very efficient.

Reference: PRL2008 Rules 17.2 and 17.18

Rule 15.2 Be as consistent as possible with iterator types.

GUIDELINE, C++ ONLY

Efficiency may decrease when iterator types are mixed because conversion may be necessary. If constant iterators are used, certain functions may not be able to be called because they only accept non-constant iterators.

```
void foo()
{
  list<int> m;
  list<int>::const_iterator first = m.begin();
  list<int>::iterator last = m.end();

  // last will need to be converted to a const_iterator, which
  // wastes time
  if( first == last )
  {
    //…
  }

  list<int> newList;

  // Incorrect, a const_iterator cannot be implicitly
  // converted to an iterator
  newList.insert(first, 5);
}
```

Reference: PRL2008 Rule 17.16

Rule 15.3 Do not rely on indirect includes of C++ Standard Library headers.

MANDATORY, C++ ONLY

Many C++ headers include other headers and some programs take advantage of this by not including some necessary headers, knowing that they're indirectly included. This is poor practice. Including all required headers helps improve code portability. Though it may be known that a certain C++ header contains another, this is not

guaranteed by the C++ standard. Thus, code that assumes this could fail to compile on another system or using a different version of the compiler.

For example, if the **vector** header includes **string** on one system, then this code will compile:

```
#include <vector>

void foo()
{
  std::vector<std::string> nameList;
  //…
}
```

However, if the code is ported to a system where **vector** does not include **string**, the code will not compile. The robust solution is to include the **string** header directly:

```
#include <string>
#include <vector>

void foo()
{
  std::vector<std::string> nameList;
  //…
}
```

Reference: PRL2008 Rule 17.20

Rule 15.4 Use an STL implementation that offers extra runtime checking or a "debug" mode.

GUIDELINE, C++ ONLY

An STL implementation that offer extra runtime checking and/or "debug" modes are sometimes referred to as "checked" implementations. Such implementations are more robust, as their versions of the STL classes perform many safety checks, such as validating iterators and verifying preconditions (e.g. arguments) of generic algorithms. These extra checks could be invaluable in finding subtle bugs in a program.

There is however a large performance penalty for all this extra checking and so the STL implementation will typically have a way to disable these checks. Presumably, they would be turned off upon release of the software.

GNU, Microsoft, and Metrowerks all offer such STL implementations.

Reference: S&A2005 Rule 83

Rule 15.5 Never place a *using* directive or *using* declaration in a header file or before an *#include* statement.

MANDATORY, C++ ONLY

The **using** directive brings in all names from a namespace (typically **std::**) while a **using** declaration brings in a specific name. If either of these commands were to appear in a header file, then everyone who included that header file would unknowingly inherit the name or names made available by the **using** statement. This is poor practice as it can cause unexpected errors in seemingly unrelated code that happens to share a header file. In addition, it is unsafe to place a **using** statement before an #**include** statement because the name or names it makes available could alter the behavior of the code in the header or header file that follows.

Implicit in this rule is that namespace qualifiers (e.g. **std::**) should be used on all namespace names in header files and before the last **#include** statement in implementation files.

Reference: S&A2005 Rule 59

Rule 15.6 Always choose a container member function over an algorithm of the same name.

MANDATORY, C++ ONLY

Container member functions frequently will be more efficient than algorithms of the same name. This is because member functions know about the internal implementation of the container and thus can be optimized to work with the container elements as efficiently as possible. An algorithm, on the other hand, has no access to the internals of the container and may be inefficient.

For example, the **find()** algorithm runs in linear time but the **find()** member function of the **map** class runs in logarithmic time. So, for the **map** class, the latter will be much faster.

Reference: PRL2008 Rule 17.19

Rule 15.7 Predicates shall be deterministic.

MANDATORY, C++ ONLY

The output of a predicate should only depend on its parameters. There is no guaranteed order of evaluation when iterating through a container, so the predicate must give the same result for an object regardless of previous calls. The predicate (*BadCounter* below) may even be copied, so there is no guarantee that the state of the predicate will be maintained. Here is an example of a predicate that violates this rule:

```
class BadCounter
{
```

```
  public:

    BadCounter() : count(0) {}
    bool operator() (const int&)
    {
      return ++count == MAX_COUNT;
    }

  private:
    const int MAX_COUNT = 1000;
    int count;
};

bool isLargeVector( std::vector<int>& vec )
{
  // Since the predicate may be copied, the count may never
  // reach MAX_COUNT no matter how many elements are in the
  // vector
  return vec.end() !=
    std::find_if(vec.begin(), vec.end(), BadCounter());
}
```

Reference: PRL2008 Rule 17.17

Rule 15.8 Do not use the STL *auto_ptr* class unless its specific memory ownership features are required.

MANDATORY, C++ ONLY

The **auto_ptr** class behaves a little differently than might be expected. Its purpose is to take *ownership* of a block of memory. To do this, the class has *destructive copy* semantics, which can create confusion if not fully understood:

```
void foo( std::auto_ptr<double> p );

int main()
{
  std::auto_ptr<double> ap( new double );
  foo( ap );

  // Now ap has been destroyed and is no longer valid.

  //…
}
```

When *ap* is passed to function *foo*, it is passed by value and thus copied to local variable *p*. This causes the memory *ap* pointed at to be destroyed. If *ap* was passed as a reference, then it would not have been destroyed.

This behavior, though useful in certain applications, is counterintuitive. Therefore, the programmer should only use **auto_ptr** with dynamically-allocated memory and

when the pointer needs ownership of a memory object. Never use **auto_ptr** where more than one pointer to the same memory at the same time is required.

Reference: PRL2008 Rule 17.21

16. Data Formats and Portability

Rule 16.1 The programmer shall assume nothing about the sizes of the integer data types, except that the range of a *char* < the range of a *short* <= the range of an *int* < the range of a *long*.

MANDATORY

The C standard defines header file **limits.h** while the C++ standard defines header **climits**. These contain symbolic names which denote the minimum and maximum values of all the integer types. These values can vary between systems but a minimum range of values for each type is defined by the standards and are guaranteed to be consistent among systems. The standards only guarantee that the range of a **char** < the range of a **short** <= the range of an **int** < the range of a **long**.

Reference: ESA2000 Rule 115

Rule 16.2 The programmer shall assume nothing about the sizes of the floating point types, except that the range of a *float* <= the range of a *double* <= the range of a *long double*.

MANDATORY

The C standard defines header file **float.h** while the C++ standard defines header **cfloat**. These contain symbolic names which denote the minimum and maximum values of all the floating point types. These values can vary between systems but a minimum range of values for each type is defined by the standards and are guaranteed to be consistent among systems. The standards only guarantee that the range of a **float** <= the range of a **double** <= the range of a **long double**.

Reference: ESA2000 Rule 116

Rule 16.3 No assumptions shall be made about how data types are represented in memory.

MANDATORY

Different systems may use different representations of data in memory, including but not limited to differences in word ordering, byte ordering within a word, bit ordering within a byte, and floating point standard (e.g. IEEE 754). Therefore, to make the program more portable, the programmer may not assume knowledge of the underlying representation.

Reference: ESA2000 Rule 117

Rule 16.4 The programmer shall assume that different data types have different representations in memory.

MANDATORY

This follows from Rule 16.3. Different data types are not guaranteed, or even expected, to share equivalent representations in memory, especially if they specify different ranges of values.

Reference: ESA2000 Rule 118

Rule 16.5 Do not assume anything about how data types are aligned in memory.

MANDATORY

The rules on how data types are aligned in memory vary among different hardware platforms so any program that makes an assumption about alignment may not be portable.

Reference: ESA2000 Rule 119

Rule 16.6 Pointers to different data types shall never be assumed to be equivalent to each other.

MANDATORY

Nothing in the C and C++ languages requires that pointers to different data types be represented in the same way. Therefore, a pointer to one type should never be assigned to a pointer to another type. Nor should a pointer to one type be cast to a pointer to another type.

EXCEPTION Pointers to **void** are considered equivalent to pointers to specific types.

EXCEPTION In certain contexts in C++, a pointer to a base class may be converted to a pointer to a derived class or vice versa.

Reference: ESA2000 Rule 120

Rule 16.7 Pointer and integer arithmetic shall not both be included in the same expression.

MANDATORY

Pointer arithmetic differs from integer arithmetic, because standard arithmetic operations applied to pointers are scaled by the size (in bytes) of the type referenced by the pointer. So incrementing a pointer with a type size of 2 bytes increases the value of the pointer by 2, and subtracting 3 from a pointer with a type size of 4 bytes would decrease the value of the pointer by 12. Due to this difference, mixing pointer and integer arithmetic in the same expression is prone to error and shall be avoided.

Here are some examples of unacceptable use:

```
int array[64];
```

```
// Improperly set the value of the second element to zero
*(array + (1 + 1)) = 0

// Improperly increment the value of the second element
*(array + 2) = *(array + 2) + 1;
```

Reference: ESA2000 Rule 121

Rule 16.8 When testing for overflow or underflow, use an unsigned data type if possible. Otherwise, try a "wider" data type.

MANDATORY

Overflow occurs when a value or an expression result is too large to be contained in the specific data type. For example, the **signed char** type in C++ spans from -128 to 127. The value 200 indicates *positive overflow* while the value -200 indicates *negative overflow*. *Underflow* occurs when a value or expression result is too small to be contained in the specific data type. More formally, underflow means that the exponent of a floating point number is a negative number whose value is too big to fit in the space allocated to hold it.

When a developer feels the need to test for overflow or underflow, one of two strategies must be employed. If the expression contains unsigned integers, then overflow can be detected without relying on the underlying hardware implementation. Simply convert any signed integers to unsigned and test for "wraparound" as in the following examples:

```
unsigned int a;
unsigned int b;
if ((a + b < a) || (a + b < b))
  // must be overflow
. . .
int c;
if ((a + static_cast<unsigned int>(c) < a) ||
    (a + static_cast<unsigned int>(c) < c))
  // must be overflow
. . .
if ((a - b > a) || a - b > b)
  // must be overflow
```

However, if the expression contains only signed integers or any floating point numbers, then there is no foolproof test for overflow or undeflow. However, many such cases can be caught by casting the values to a relatively "wider" type and checking against the maximum possible value of the original type. Generally speaking, the widest type available is the best choice. Here is an example for overflow:

```
#include <climits>
short s;
```

```
short t;
if (static_cast<long>(s) * static_cast<long>(t) > SHRT_MAX)
  // must be overflow
```

Underflow might be detected like this:

```
#include <cfloat>
float f;
float g;
if (static_cast<long double>(f) *
    static_cast<long double>(g) < FLT_MIN)
  // must be underflow
```

Reference: ESA2000 Rule 122

Rule 16.9 When assigning values from a relatively wider data type to shorter data type, no unintended loss of precision shall occur.

MANDATORY

When converting from any relatively wider data type to a shorter data type, the programmer shall be careful to consider the implications of any loss of precision that may occur. For example, when converting from a **float** to an **int**, the programmer must ensure that the converted value that has been truncated will not adversely affect the program or cause undefined behavior. If necessary, a comment shall be added to explain why a loss of precision is acceptable during such a conversion.

NOTE Some compilers have options that turn on checking for the situations described in this rule and these options should be used whenever practical.

Reference: ESA2000 Rule 123

17. Memory Management

Rule 17.1 The success or failure of dynamic memory allocation functions shall always be checked.

MANDATORY

Functions such as **malloc()**, **realloc()**, and **calloc()**, which dynamically allocate memory, can fail if there is not enough system memory available or for other reasons. These functions return a **NULL** (0 value) pointer if they fail. The return value should always be validated to ensure there wasn't a problem allocating memory.

Note that Rule 17.6 addresses verification of C++ style allocations.

Example:

```
char* buffer;
int size = 20;
buffer = static_cast<char*>(malloc(size));

if( buffer == 0 )
{
  // The memory could not be allocated
  // Print an error message and decide what to do

  // Most programs will need to exit
  exit(1);
}

free(buffer);
exit 0;
```

Reference: ESA2000 Rule 99

Rule 17.2 The *free()* function shall be used to return dynamic memory allocated by *malloc()*, *calloc()*, or *realloc()* to the system.

MANDATORY

The **free()** function will return memory to the system that was allocated by the **malloc()** family of functions. If such memory is not deallocated using **free()**, it will not be returned to the system until the program exits. This can cause memory leaks, where the program's memory space grows larger and larger. This can be especially damaging if the program is meant to run for a long time.

Here is an example:

```
char* buffer;
buffer = static_cast<char*>(malloc(50));
```

```
if( buffer == 0 )
{
  // Exit the program if the memory
  // couldn't be allocated
  exit(1);
}

// Use the buffer
//…

// When finished with the memory, free it
free(buffer);
exit 0;
```

Note that **free()** must only be used for memory allocated with **malloc()**, **realloc()**, or **calloc()**. It must never be used for memory allocated with the **new** operator. Similarly, the **delete** operator must only be used for memory allocated with the **new** operator, not for memory allocated with **malloc()**, **realloc()**, or **calloc()**.

Reference: ESA2000 Rule 100

Rule 17.3 When using *realloc()*, the programmer shall retain the original pointer to dynamic memory in case the call fails.

MANDATORY

The **realloc()** function is meant to resize a block of memory that was previously allocated using **malloc()**, **calloc()**, or **realloc()**. It normally allocates a new block of memory of the requested size and frees the old block. However, if the allocation of the new memory fails, a memory leak could be introduced even if the return value is checked.

Here is an example of the wrong way to use **realloc()**:

```
char* buffer = static_cast<char*>(malloc(20));
if( buffer == 0 )
{
  exit(1);
}

// Now we want to increase the size of buffer
buffer = static_cast<char*>(realloc(buffer, 100));

if( buffer == 0 )
{
  // Try to recover from error
}

exit 0;
```

In the above example, there are two allocations: the first one of 20 bytes and the second one of 100 bytes. If the allocations succeed, then the call to **realloc()** frees the previously allocated block of 20 bytes, and buffer now points to a block of 100 bytes. However, if the **realloc()** call fails, it will return a **NULL** pointer in variable *buffer*. But now, we do not have a pointer to the previously allocated block of 20 bytes nor was it freed by **realloc()**. Thus, we have a memory leak.

This example shows the correct way to use **realloc()**:

```
char* buffer = static_cast<char*>(malloc(20));
if( buffer == 0 )
{
  exit(1);
}

// Now we want to increase the size of buffer
char* tmp = static_cast<char*>(realloc(buffer, 100));

// Check to make sure allocation succeeded
if( tmp != 0 )
{
  // Set buffer to point to the new memory
  buffer = tmp;
}
else
{
  // The allocation failed, but we still have a pointer to the
  // first buffer

  // Try to recover from error
}

exit 0;
```

In the above example, the return value of **realloc()** is checked before *buffer* is set to point to the new block. If the **realloc()** failed, we still have a pointer to the previously allocated block and can decide what to do.

Reference: ESA2000 Rule 101

Rule 17.4 Memory shall be managed using the *new operator* and *delete operator* rather than *malloc()*, *realloc()*, and *free()*.

MANDATORY, C++ ONLY

The C++ **new** and **delete** operators are preferred over **malloc()**, **realloc()**, and **free()** because they call constructors and destructors for objects instead of allocating raw memory. Also, the **new** operator automatically calculates the size of the block to be allocated, while **malloc()** and **realloc()** require the programmer to calculate the exact number of bytes needed.

Reference: ESA2000 Rule 102

Rule 17.5 When memory is allocated using the *new[] operator*, it shall be deleted using the *delete[] operator*.

MANDATORY, C++ ONLY

The **new[]** operator is used to allocate an array of objects (it is meant to replace **calloc()**). Memory that is allocated with **new[]** must be deleted by using **delete[]**. Since there is no difference between a pointer to an object and a pointer to an array of objects, if the **delete** operator is used instead of **delete[]**, a memory leak will occur because **delete** frees a single object, not the entire array.

For example:

```
class Obj1
{
  //…
};

int main()
{
  // Allocate an array of 10 Obj1 objects
  Obj1[] objArray = new Obj1[10];

  // Use array
  //…

  // WRONG WAY TO DELETE ARRAY
  delete objArray;

  // Correct way to delete array
  delete[] objArray;

  return 0;
}
```

Reference: ESA2000 Rule 103

Rule 17.6 A user-defined function to handle memory allocation errors must be specified by *set_new_handler()* in every program which does dynamic memory allocation.

MANDATORY, C++ ONLY

If an allocation using the **new** or **new[]** operator fails, the function specified using **set_new_handler()** is called. This function can try to free up some memory and return, in which case the **new** operator will retry the allocation. If no memory can be returned to the system, the function should throw a **bad_alloc** exception or abort the program. This function should only return if memory was freed, because when it returns, the **new** operator will attempt the allocation again. If no memory is freed, the handler function may be called over and over indefinitely.

Reference: ESA2000 Rule 104

18. Miscellaneous Language Rules

Rule 18.1 The use of global entities shall be avoided.

MANDATORY

Global entities, such as variables, constants, and types, should be used rarely, if at all.

In C++, all constants, enumerated types, type definitions, structures, variables, and functions should be contained in the most relevant class. If a global entity must be used, then it should be (a) part of a named namespace, and (b) always accessed through an explicit namespace reference (e.g. using the "::" notation).

The use of global entities should be minimized in C. However, it is recognized that they are sometimes necessary. In such cases, each entity's name shall contain a prefix, which denotes which subsystem or library contains the definition of that entity.

Reference: ESA2000 Rule 36

Rule 18.2 Either a tag name or a forward declaration shall be used for a self-referential structure.

MANDATORY

Using a tag is more compact because the declaration appears in only one place. However, two names for the structure must be used. For example:

```
typedef struct tree
{
  int data;
  struct tree* leftSubTree;
  struct tree* rightSubTree;
} tree_t;
```

In this case, *tree_t* and *struct tree* both refer to the structure.

Forward declaration is slightly more complicated because the declaration is separate from the definition of the structure. However, this method allows the details of the structure to be in the implementation file and only the declaration in the interface file, if desired. Example:

```
typedef struct tree tree;

struct tree
{
  int data;
  tree* leftSubTree;
  tree* rightSubTree;
};
```

Reference: ESA2000 Rule 57

Rule 18.3 When possible, use C++ templates to generalize and reuse code.

GUIDELINE, C++ ONLY

C++ templates provide a convenient way to reuse code, which reduces the length of the code, and hopefully makes the code easier to understand and maintain. If a function that is replicated for several different input types needs to be updated, the changes will need to be replicated in all versions of the function. But if the function were templatized, the changes would only need to be made in one place.

Reference: ESA2000 Rule 86

Rule 18.4 The programmer shall not overload the following operators: *logical and* ("&&"), *logical or* ("||"), *sequential evaluation* (","), and *conditional* ("?:").

MANDATORY, C++ ONLY

These four operators are *sequence point* operators as described in Rule 14.2. They cause expressions and statements in a program to be evaluated in strictly-defined ways and orders. By overloading these operators, they will lose their ability to function as sequence points. This means the code using them is not guaranteed to execute in exactly the same way as other parts of the code that do not have access to them.

Reference: ESA2000 Rule 94

Rule 18.5 Check all return values of library functions for errors.

GUIDELINE

The programmer should never assume that library functions will always succeed. Thus, the return values of such functions should be checked for error codes.

There are however some functions that return a status code but rarely fail under normal conditions, such as **sprintf()** and **fclose()**. For such functions, it is the common custom not to check the return value; this is why this rule is categorized as a guideline only.

Reference: ESA2000 Rule 106

Rule 18.6 Add diagnostic code to check for conditions that "should never happen".

GUIDELINE

In certain situations, it is worthwhile to check for unusual cases which the programmer believes should never happen. Because all the conditions under which the software will be used cannot be anticipated, it is not out of the question that some of these cases may actually occur over time. Examples might include adding diagnostic code to the **default** case of a **switch** statement or to a dangling **else** statement, even though neither spot is expected to ever be reached when the

program is running. Such diagnostics may perhaps not execute for years, but, if they finally do, they can save those maintaining the software tremendous amounts of debugging time.

Reference: ESA2000 Rule 107

19. Sample File Format Templates

For Rule 19.1, Rule 19.2, and Rule 19.3, **$RCSfile$**, **$Revision$**, and **$Date$** are CVS keywords that will automatically be expanded by CVS. If another version control system is used, these keywords may be replaced by analogous terms appropriate for that system.

Similarly, in Rule 19.1 through Rule 19.5, the Doxygen markup terms **@file**, **@note**, **@class**, **@param**, **@pre**, **@post**, and **@return** are used. If a different tool is used to extract documentation from the source code, this markup may be replaced by analogous markup understood by that tool.

Rule 19.1 Use the following template for the header of a C++ implementation file not related to any specific class.

MANDATORY, C++ ONLY

The standard header section for a C++ implementation file is shown below. A single blank line separates the optional copyright notice from the first Doxygen comment block.

```
//===========================================================================
//
//   <Optional copyright statement; delete if not used.  If included, use:>
//   (c) Copyright, <<Insert Year>> <<Insert Organization>>.
//   The Government shall have unlimited use of this source code and
//   documentation for government purposes.
//
//   File: $RCSfile$
//   Version: $Revision$  Dated: $Date$
//
//===========================================================================

/**
 *
 *  @file <filename>
 *
 *  <file description>
 *
 *  @note
 *
 *  <Optional.  Delete if not used>
 *
 */
```

Rule 19.2 Use the following template for the header of a C++ implementation file for a given class.

MANDATORY, C++ ONLY

When documenting a C++ implementation file for a given class (typically the one with the constructors in it), the following variant of the file header in Rule 19.1 with

the Doxygen **@class** keyword should be used. This is necessary in order for Doxygen to associate the subsequent description with the class.

```
//===========================================================================
//
//   <Optional copyright statement; delete if not used.  If included, use:>
//   (c) Copyright, <<Insert Year>> <<Insert Organization>>.
//   The Government shall have unlimited use of this source code and
//   documentation for government purposes.
//
//   File: $RCSfile$
//   Version: $Revision$  Dated: $Date$
//
//===========================================================================

/**
 *
 *  @file <filename>
 *
 *  <file description>
 *
 *  @class <classname>
 *
 *  <class description>
 *
 *  @note
 *
 *  <Optional.  Delete if not used>
 *
 */
```

Rule 19.3 Use the following template for the header of a C++ interface file.

MANDATORY, C++ ONLY

The standard template for a C++ interface file is shown below. A single blank line separates the optional copyright notice from the first Doxygen comment block. Preprocessor file guards follow (Rule 3.6).

```
//===========================================================================
//
//   <Optional copyright statement; delete if not used.  If included, use:>
//   (c) Copyright, <<Insert Year>> <<Insert Organization>>.
//   The Government shall have unlimited use of this source code and
//   documentation for government purposes.
//
//   File: $RCSfile$
//   Version: $Revision$  Dated: $Date$
//
//===========================================================================

/**
 *
 *  @file <filename.h>
 *
```

```
 *  <brief file description>
 *
 */

#ifndef FILENAME_H
#define FILENAME_H

<body>

#endif
```

Rule 19.4 Use the following template for a class declaration.

MANDATORY, C++ ONLY

The standard template for a class declaration is shown below.

```
class SampleClass
{
  public:

    /**
     *  <brief constructor description>
     *
     *  <full multi-line description>
     *
     *  @param[in]  <paramname1>   <Description>
     *  @param[in]  <paramname2>   <Description>
     *
     *  @pre  <description. Delete if not used>
     *
     *  @note  <brief description>   [optional]
     *
     *  <Extended note description>
     *
     */

    SampleClass(
      <paramtype1> <paramname1>
      <paramtype2> <paramname2>
    );

    /**
     *  <brief destructor description>
     *
     *  <full multi-line description>
     *
     *  @param[in]  <paramname1>   <Description>
     *
     *  @note  <brief description>   [optional]
     *
     *  <Extended note description>
     *
     */

    ~SampleClass(
```

```
      <paramtype1> <paramname1>
    );

    /**
     *  <brief member function description>
     *
     *  <full multi-line description>
     *
     *  @param[in]      <paramname1>   <Description>
     *  @param[out]     <paramname2>   <Description>
     *  @param[in,out]  <paramname3>   <Description>
     *
     *  @pre  <description. Delete if not used>
     *
     *  @return  <description>   [Should indicate "None" for void functions]
     *
     *  @post  <description. Delete if not used>
     *
     *  @note  <brief description>   [optional]
     *
     *  <Extended note description>
     *
     */

    returntype memberFunction(
      <paramtype1> <paramname1>
      <paramtype2> <paramname2>
      <paramtype3> <paramname3>
    );

  protected:

  private:

    <membertype> member1; /**< member description */

    // An alternative format is the following
    /** member description */
    <membertype> member1;

};
```

An acceptable alternative to the C-style Doxygen parameter comment

```
/**< description */
```

is the C++-style comment

```
///< description
```

Rule 19.5 Use the following template for a function header in an interface file.

MANDATORY, C++ ONLY

The standard template for a function declaration in an interface file is shown below. Interface-related Doxygen comments shall be placed in the interface file while

implementation-related Doxygen comments (if any) shall be placed in the implementation file.

```
/**
 *  <brief description>
 *
 *  <full multi-line description>
 *
 *  @param[in]       <paramname1>   <Description>
 *  @param[out]      <paramname2>   <Description>
 *  @param[in,out]   <paramname3>   <Description>
 *
 *  @pre  <description. Delete if not used>
 *
 *  @return  <description>   [Should indicate "None" for void functions]
 *
 *  @post  <description. Delete if not used>
 *
 *  @note  <brief description>   [optional]
 *
 *  <Extended note description>
 *
 */

Functionname(
  <paramname1>
  <paramname2>
  <paramname3>
);
```

20. Glossary

ANSI	American National Standards Institute.
accessor	Class function which returns the value of a specific member of the class; colloquially called a *get function* or *getter*.
base class	A class from which one or more other classes are derived; also referred to as a *superclass* or *parent class*.
CM	Configuration management; the managing of software source code , both on the file level and the release level.
CVS	Concurrent Versions System; a software version control system. `http://www.ximbiot.com/cvs`
derived class	A class that is built upon another class, inheriting some of its members and functionality from that class; also referred to as a *subclass* or *child class*.
Doxygen	An inline markup language and documentation generator for C, C++, and other languages. `http://www.doxygen.org`
grep	Unix command that does text searches on a file or files.
ISO	International Standards Organization.
lifetime	The temporal period (or periods) during the life of the program when a variable or other declared name exists; also known as *extent*; contrast with *scope*.
make	Unix command that can be configured to automatically build (compile and link) programs and libraries.
mutator	Class function which sets or modifies a specific member of the class; colloquially called a *set function* or *setter*.
STL	The C++ Standard Template Library.
scope	The physical area of a program where a variable or other declared name is known; contrast with *lifetime*.
URL	Uniform Resource Locator; an address of a Web page or some other Internet resource.

21. References

[ESA2000] *C and C++ Coding Standards*, European Space Agency Board for Software Standardisation and Control (ESA BSSC), Paris, France, 2000-03-30.

[K&R1998] *The C Programming Language*, Second Edition, Brian W. Kernigham and Dennis M. Ritchie, Prentice Hall, Englewood Cliffs, NJ, 1988.

C A Reference Manual, Fifth Edition, Samuel P. Harbison III and Guy L. Steele Jr., Prentice Hall, Upper Saddle River, NJ, 2002.

[S&A2005] *C++ Coding Standards*, Herb Sutter and Andrei Alexandrescu, Addison-Wesley, Boston, MA, 2005-02.

C++ In a Nutshell, Ray Lischner, O'Reilly Media, Sebastopol, CA, 2003-05.

Effective STL, Scott Meyers, Addison-Wesley, 2001.

[PRL2008] *High Integrity C++ Coding Standard Manual*, Version 3.2, Programming Research Limited, Surrey, Great Britain, 2008-10-03.

Thinking in C++, 2nd Edition, Volume 1: Introduction to Standard C++, Bruce Eckel, Prentice Hall, Upper Saddle River, NJ, 2000.
`http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html`

Thinking in C++, Volume 2: Practical Programming, Bruce Eckel and Chuck Allison, Pearson Prentice Hall, Upper Saddle River, NJ, 2004.
`http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html`