JOINT STRIKE FIGHTER

AIR VEHICLE

C++ CODING STANDARDS

FOR THE SYSTEM DEVELOPMENT AND DEMONSTRATION PROGRAM

Document Number 2RDU00001 Rev C

December 2005

Copyright 2005 by Lockheed Martin Corporation.

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

This page intentionally left blank

TABLE OF CONTENTS

Table 1. Change Log

Revision ID	Document Date	Change Authority	Affected Paragraphs	Comments
0001 Rev B	Oct 2005	K. Carroll	All	Original
0001 Rev C	Nov 2005	K. Carroll	Change log - Added	Add change log.
			Section 1, point 3 Rule 52 Rule 76 Rule 91 Rule 93 Rule 129 Rule 167 Rule 218 Appendix A, Rule 3 Table 2	Corrected spelling errors.
			Rule 159 - clarify that "unary &" is intended.	Both binary and unary forms of "&" exist. Clarification is added to specify that the rule is concerned with the unary form.
			Rule 32 - clarification of the scope of the rule. Also, example added in appendix for rule 32.	The rule does not apply to a particular partitioning of template classes and functions.

1 INTRODUCTION

The intent of this document is to provide direction and guidance to C++ programmers that will enable them to employ good programming style and proven programming practices leading to safe, reliable, testable, and maintainable code. Consequently, the rules contained in this document are required for Air Vehicle C++ development[1] and recommended for non-Air Vehicle C++ development.

As indicated above, portions of Air Vehicle (AV) code will be developed in C++. C++ was designed to support data abstraction, object-oriented programming, and generic programming while retaining compatibility with traditional C programming techniques. For this reason, the AV Coding Standards will focus on the following:

1. Motor Industry Software Reliability Association (MISRA) Guidelines For The Use Of The C Language In Vehicle Based Software,

2. Vehicle Systems Safety Critical Coding Standards for C, and

3. C++ language-specific guidelines and standards.

The MISRA Guidelines were written specifically for use in systems that contain a safety aspect to them. The guidelines address potentially unsafe C language features, and provide programming rules to avoid those pitfalls. The Vehicle Systems Safety Critical Coding Standards for C, which are based on the MISRA C subset, provide a more comprehensive set of language restrictions that are applied uniformly across Vehicle Systems safety critical applications. The AV Coding Standards build on the relevant portions of the previous two documents with an additional set of rules specific to the appropriate use C++ language features (e.g. inheritance, templates, namespaces, etc.) in safety-critical environments.

Overall, the philosophy embodied by the rule set is essentially an extension of C++'s philosophy with respect to C. That is, by providing "safer" alternatives to "unsafe" facilities, known problems with low-level features are avoided. In essence, programs are written in a "safer" subset of a superset.

[1] TBD: Required for Air Vehicle non-prime teams?

2 REFERENCED DOCUMENTS

1. ANSI/IEEE Std 754, IEEE Standard for Binary Floating-Point Arithmetic, 1985.

2. Bjarne Stroustrup. *The C++ Programming Language, 3rd Edition*. Addison-Wesley, 2000.

3. Bjarne Stroustrup. *Bjarne Stroustrup's C++ Glossary*.

4. Bjarne Stroustrup. *Bjarne Stroustrup's C++ Style and Technique FAQ*.

5. Barbara Liskov. *Data Abstraction and Hierarchy, SIGPLAN Notices*, 23, 5 (May, 1988).

6. Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Design, 2nd Edition.* Addison-Wesley, 1998.

7. Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs.* Addison-Wesley, 1996.

8. Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Vehicle Based Software,* April 1998.

9. ISO/IEC 10646-1, Information technology - Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane, 1993.

10. ISO/IEC 14882:2003(E), *Programming Languages – C++*. American National Standards Institute, New York, New York 10036, 2003.

11. ISO/IEC 9899: 1990, *Programming languages - C*, ISO, 1990.

12. JSF Mission Systems Software Development Plan.

13. JSF System Safety Program Plan. DOC. No. 2YZA00045-0002.

14. *Programming in C++ Rules and Recommendations*.
    Copyright © by Ellemtel Telecommunication Systems Laboratories
    Box 1505, 125 25 Alvsjo, Sweden
    Document: M 90 0118 Uen, Rev. C, 27 April 1992.

    Used with permission supplied via the following statement:

    Permission is granted to any individual or institution to use, copy, modify and distribute this document, provided that this complete copyright and permission notice is maintained intact in all copies.

15. RTCA/DO-178B, *Software Considerations in Airborne Systems and Equipment Certification*, December 1992.

3 GENERAL DESIGN

This coding standards document is intended to help programmers develop code that conforms to safety-critical software principles, i.e., code that does not contain defects that could lead to catastrophic failures resulting in significant harm to individuals and/or equipment. In general, the code produced should exhibit the following important qualities:

Reliability: Executable code should consistently fulfill all requirements in a predictable manner.

Portability: Source code should be portable (i.e. not compiler or linker dependent).

Maintainability: Source code should be written in a manner that is consistent, readable, simple in design, and easy to debug.

Testability: Source code should be written to facilitate testability. Minimizing the following characteristics for each software module will facilitate a more testable and maintainable module:

1. code size
2. complexity
3. static path count (number of paths through a piece of code)

Reusability: The design of reusable components is encouraged. Component reuse can eliminate redundant development and test activities (i.e. reduce costs).

Extensibility: Requirements are expected to evolve over the life of a product. Thus, a system should be developed in an extensible manner (i.e. perturbations in requirements may be managed through local extensions rather than wholesale modifications).

Readability: Source code should be written in a manner that is easy to read, understand and comprehend.

Note that following the guidelines contained within this document will not guarantee the production of an error-free, *safe* product. However, adherence to these guidelines, as well as the processes defined in the Software Development Plan [12], will help programmers produce clean designs that minimize common sources of mistakes and errors.

3.1 Coupling & Cohesion

Coupling and cohesion are properties of a system that has been decomposed into modules. Cohesion is a measure of how well the parts in the same module fit together. Coupling is a measure of the amount of interaction between the different modules in a system. Thus, cohesion deals with the elements within a module (how well-suited elements are to be part of the same module) while coupling deals with the relationships among modules (how tightly modules are glued together).

Object-oriented design and implementation generally support desirable coupling and cohesion characteristics. The design principles behind OO techniques lead to data cohesion within modules. Clean interfaces between modules enable the modules to be loosely coupled. Moreover, data encapsulation and data protection mechanisms provide a means to help enforce the coupling and cohesion goals.

Source code should be developed as a set of modules as loosely coupled as is reasonably feasible. Note that generic programming (which requires the use of templates) allows source code to be written with loose coupling and without runtime overhead.

Examples of tightly coupled software would include the following:

- many functions tied closely to hardware or other external software sources, and
- many functions accessing global data.

There may be times where tightly coupled software is unavoidable, but its use should be both minimized and localized as suggested by the following guidelines:

- limit hardware and external software interfaces to a small number of functions,
- minimize the use of global data, and
- minimize the exposure of implementation details.

3.2 Code Size and Complexity

AV Rule 1

Any one function (or method) **will** contain no more than 200 logical source lines of code (L-SLOCs).

```
For this unit and later, no routine is to be larger
than one screen.
```

Rationale: Long functions tend to be complex and therefore difficult to comprehend and test.

Note: Section 4.2.1 defines **should** and **shall** rules as well the conditions under which deviations from **should** or **shall** rules are allowed.

AV Rule 2

There **shall not** be any self-modifying code.

Rationale: Self-modifying code is error-prone as well as difficult to read, test, and maintain.

AV Rule 3

All functions **shall** have a cyclomatic complexity number of 20 or less.

Rationale: Limit function complexity. See AV Rule 3 in Appendix A for additional details.

Exception: A function containing a switch statement with many case labels may exceed this limit.

Note: Section 4.2.1 defines **should** and **shall** rules as well the conditions under which deviations from **should** or **shall** rules are allowed.

4 C++ CODING STANDARDS

4.1 Introduction

The purpose of the following rules and recommendations is to define a C++ programming style that will enable programmers to produce code that is more:

- correct,
- reliable, and
- maintainable.

In order to achieve these goals, programs should:

- have a consistent style,
- be portable to other architectures,
- be free of common types of errors, and
- be understandable, and hence maintainable, by different programmers.

4.2 Rules

4.2.1 Should, Will, and Shall Rules

There are three types of rules: **should, will,** and **shall** rules. Each rule contains either a **"should", "will"** or a "**shall"** in bold letters indicating its type.

- **Should** rules are advisory rules. They strongly suggest the recommended way of doing things.
- **Will** rules are intended to be mandatory requirements. It is expected that they will be followed, but they do not require verification. They are limited to non-safety-critical requirements that cannot be easily verified (e.g., naming conventions).
- **Shall** rules are mandatory requirements. They must be followed and they require verification (either automatic or manual).

4.2.2 Breaking Rules

AV Rule 4

To break a **"should"** rule, the following approval must be received by the developer:

- approval from the software engineering lead (obtained by the unit approval in the developmental CM tool)

AV Rule 5

To break a **"will"** or a **"shall"** rule, the following approvals must be received by the developer:

- approval from the software engineering lead (obtained by the unit approval in the developmental CM tool)
- approval from the software product manager (obtained by the unit approval in the developmental CM tool)

AV Rule 6

Each deviation from a **"shall"** rule **shall** be documented in the file that contains the deviation). Deviations from this rule **shall not** be allowed, AV Rule 5 notwithstanding.

4.2.3 Exceptions to Rules

Some rules may contain **exceptions**. If a rule does contain an exception, then approval is not required for a deviation allowed by that exception

AV Rule 7

Approval **will not** be required for a deviation from a **"shall"** or **"will"** rule that complies with an **exception** specified by that rule.

4.3 Terminology

1. An **abstract base class** is a class from which no objects may be created; it is only used as a base class for the derivation of other classes. A class is abstract if it includes at least one member function that is declared as pure virtual.

2. An **abstract data type** is a type whose internal form is hidden behind a set of access functions. Objects of the type are created and inspected only by calls to the access functions. This allows the implementation of the type to be changed without requiring any changes outside the module in which it is defined.

3. An **accessor** function is a function which returns the value of a data member.

4. A **catch clause** is code that is executed when an exception of a given type is raised. The definition of an exception handler begins with the keyword catch.

5. A **class** is a user-defined data type which consists of data elements and functions which operate on that data. In C++, this may be declared as a class; it may also be declared as a struct or a union. Data defined in a class is called member data and functions defined in a class are called member functions.

6. A **class template** defines a family of classes. A new class may be created from a class template by providing values for a number of arguments. These values may be names of types or constant expressions.

7. A **compilation unit** is the source code (after preprocessing) that is submitted to a compiler for compilation (including syntax checking).

8. A **concrete type** is a type without virtual functions, so that objects of the type can be allocated on the stack and manipulated directly (without a need to use pointers or references to allow the possibility for derived classes). Often, small self-contained classes. [3]

9. A **constant member function** is a function which may not modify data members.

10. A **constructor** is a function which initializes an object.

11. A **copy constructor** is a constructor in which the first argument is a reference to an object that has the same type as the object to be initialized.

12. **Dead code** is "executable object code (or data) which, as a result of a design error cannot be executed (code) or used (data) in an operational configuration of the target computer environment and is not traceable to a system or software requirement." [9]

13. A **declaration** of a variable or function announces the properties of the variable or function; it consists of a type name and then the variable or function name. For functions, it tells the compiler the name, return type and parameters. For variables, it tells the compiler the name and type.

```
int32   fahr;
int32   foo ();
```

14. A **default constructor** is a constructor which needs no arguments.

15. A **definition** of a function tells the compiler how the function works. It shows what instructions are executed for the function.

```
int32   foo ()
{
   // Statements
}
```

16. An **enumeration type** is an explicitly declared set of symbolic integer constants. In C++ it is declared as an enum.

17. An **exception** is a run-time program anomaly that is detected in a function or member function. Exception handling provides for the uniform management of exceptions.

18. A **forwarding function** is a function which does nothing more than call another function.

19. A **function template** defines a family of functions. A new function may be created from a function template by providing values for a number of arguments. These values may be names of types or constant expressions.

20. An **identifier** is a name which is used to refer to a variable, constant, function or type in C++. When necessary, an identifier may have an internal structure which consists of a prefix, a name, and a suffix (in that order).

21. An **iterator** is an object that con be used to traverse a data structure.

22. A **macro** is a name for a text string which is defined in a *#define* statement. When this name appears in source code, the compiler replaces it with the defined text string.

23. **Multiple inheritance** is the derivation of a new class from more than one base class.

24. A **mutator** function is a function which sets the value of a data member.

25. The **one definition rule** - there must be exactly one definition of each entity in a program. If more than one definition appears, say because of replication through header files, the meaning of all such duplicates must be identical. [3]

26. An **overloaded function name** is a name which is used for two or more functions or member functions having different argument types.

27. An **overridden member function** is a member function in a base class which is re-defined in a derived class.

28. A **built-in data type** is a type which is defined in the language itself, such as int.

29. **Protected members** of a class are member data and member functions which are accessible by specifying the name within member functions of derived classes.

30. **Public members** of a class are member data and member functions which are accessible everywhere by specifying an instance of the class and the name.

31. A **pure virtual function** is one with an initializer = 0 in its declaration. Making a virtual function pure makes the class abstract. A pure virtual function must be overridden in at least one derived class.

32. A **reference** is another name for a given variable. In C++, the 'address of' (&) operator is used immediately after the data type to indicate that the declared variable, constant, or function argument is a reference.

33. The **scope** of a name refers to the context in which it is visible. [Context, here, means the functions or blocks in which a given variable name can be used.]

34. A **side effect** is the change of a variable as a by-product of an evaluation of an expression.

35. A **structure** is a user-defined type for which all members are public by default.

36. A **typedef** is another name for a data type, specified in C++ using a typedef declaration.

37. **Unqualified type** is a type that does not have const or volatile as a qualifier.

38. A **user-defined data type** is a type which is defined by a programmer in a class, struct, union, or enum definition or as an instantiation of a class template.

4.4 Environment

4.4.1 Language

AV Rule 8

All code **shall** conform to ISO/IEC 14882:2002(E) standard C++. [10]

Rationale: ISO/IEC 14882 is the international standard that defines the C++ programming language. Thus all code shall be well-defined with respect to ISO/IEC 14882. Any language extensions or variations from ISO/IEC 14882 shall not be allowed.

4.4.2 Character Sets

Note that the rules in this section may need to be modified if one or more foreign languages will be used for input/output purposes (e.g. displaying information to pilots).

AV Rule 9 (MISRA Rule 5, Revised)

Only those characters specified in the C++ basic source character set **will** be used. This set includes 96 characters: the space character, the control characters representing horizontal tab, vertical tab, form feed, and newline, and the following 91 graphical characters:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
_ { } [ ] # ( ) < > % : ; . ? * + -
/ ^ & | ~ ! = , \ " '
```

Rationale: Minimal required character set.

AV Rule 10 (MISRA Rule 6)

Values of character types **will** be restricted to a defined and documented subset of ISO 10646-1. [9]

Rationale: 10646-1 represents an international standard for character mapping. For the basic source character set, the 10646-1 mapping corresponds to the ASCII mapping.

AV Rule 11 (MISRA Rule 7)

Trigraphs **will not** be used.

Trigraph sequences are three-character sequences that are replaced by a corresponding single character, as follows:

Alternative	Primary	alternative	primary	alternative	primary
??=	#	??(	[	??<	{
??/	\	??)	]	??>	}
??'	^	??!	|	??-	~

The trigraph sequences provide a way to specify characters that are missing on some terminals, but that the C++ language uses.

Rationale: Readability. See AV Rule 11 in Appendix A.

Note: trigraphs can often be disabled via compiler flags (e.g.`-no_alternative_tokens` for the Green Hills C/C++ compiler suite)

AV Rule 12 (Extension of MISRA Rule 7)

The following digraphs **will not** be used:

Alternative	Primary	alternative	Primary
<%	{	:>	]
%>	}	%:	#
<:	[	%:%:	##

The digraphs listed above provide a way to specify characters that are missing on some terminals, but that the C++ language uses.

Rationale: Readability. See AV Rule 12 in Appendix A.

Note: Digraphs can often be disabled via compiler flags (e.g.`-no_alternative_tokens` for the Green Hills C/C++ compiler suite)

AV Rule 13 (MISRA Rule 8)

Multi-byte characters and wide string literals **will not** be used.

Rationale: Both multi-byte and wide characters may be composed of more than one byte. However, certain aspects of the behavior of multi-byte characters are implementation-defined. [10]

AV Rule 14

Literal suffixes **shall** use uppercase rather than lowercase letters.

Rationale: Readability.

Example:

```
const int64     fs_frame_rate = 64l;    // Wrong!  Looks too much like 641
const int64     fs_frame_rate = 64L;    // Okay
```

4.4.3 Run-Time Checks

AV Rule 15 (MISRA Rule 4, Revised)

Provision **shall** be made for run-time checking (defensive programming).

Rationale: For SEAL 1 or SEAL 2 software [13], provisions shall be made to ensure the proper operation of software and system function. See AV Rule 15 in Appendix A for additional details.

4.5 Libraries

AV Rule 16

Only DO-178B level A [15] certifiable or SEAL 1 C/C++ libraries **shall** be used with safety-critical (i.e. SEAL 1) code [13].

Rationale: Safety.

Note: All libraries used must be DO-178B level A certifiable or written in house and developed using the same software development processes required for all other safety-critical software. This includes both the run-time library functions as well as the C/C++ standard library functions. [10,11] Note that we expect certifiable versions of the C++ standard libraries to be available at some point in the future. These certifiable libraries would be allowed under this rule.

4.5.1 Standard Libraries

AV Rule 17 through **AV Rule 25** prohibit the use of a number of features whose behaviors are local-specific, unspecified, undefined, implementation-defined, or otherwise poorly defined and hence error prone.

AV Rule 17 (MISRA Rule 119)

The error indicator *errno* **shall not** be used.

Exception: If there is no other reasonable way to communicate an error condition to an application, then *errno* may be used. For example, third party math libraries will often make use of *errno* to inform an application of underflow/overflow or out-of-range/domain conditions. Even in this case, *errno* should only be used if its design and implementation are well-defined and documented.

AV Rule 18 (MISRA Rule 120)

The macro *offsetof,* in library <stddef.h>, **shall not** be used.

AV Rule 19 (MISRA Rule 121)

<locale.h> and the *setlocale* function **shall not** be used.

AV Rule 20 (MISRA Rule 122)

The *setjmp* macro and the *longjmp* function **shall not** be used.

AV Rule 21 (MISRA Rule 123)

The signal handling facilities of <signal.h> **shall not** be used.

AV Rule 22 (MISRA Rule 124, Revised)

The input/output library <stdio.h> **shall not** be used.

AV Rule 23 (MISRA Rule 125)

The library functions *atof, atoi* and *atol* from library <stdlib.h> **shall not** be used.

Exception: If required, *atof, atoi* and *atol* may be used only after design and implementation are well-defined and documented, especially in regards to precision and failures in string conversion attempts.

AV Rule 24 (MISRA Rule 126)

The library functions *abort, exit, getenv* and *system* from library <stdlib.h> **shall not** be used.

AV Rule 25 (MISRA Rule 127)

The time handling functions of library <time.h> **shall not** be used.

4.6 Pre-Processing Directives

Since the pre-processor knows nothing about C++, it should not be used to do what can otherwise be done in C++.

AV Rule 26

Only the following pre-processor directives **shall** be used:

1. #ifndef
2. #define
3. #endif
4. #include

Rationale: Limit the use of the pre-processor to those cases where it is necessary.

Note: Allowable uses of these directives are specified in the following rules.

4.6.1 #ifndef and #endif Pre-Processing Directives

AV Rule 27

#ifndef, #define and #endif **will** be used to prevent multiple inclusions of the same header file. Other techniques to prevent the multiple inclusions of header files **will not be** used.

Rationale: Eliminate multiple inclusions of the same header file in a standard way.

Example: For SomeHeaderFileName.h

```
#ifndef     Header_filename
#define     Header_filename

    // Header declarations…

#endif
```

AV Rule 28

The #ifndef and #endif pre-processor directives **will** only be used as defined in AV Rule 27 to prevent multiple inclusions of the same header file.

Rationale: Conditional code compilation should be kept to a minimum as it can significantly obscure testing and maintenance efforts.

4.6.2 *#define* Pre-Processing Directive

AV Rule 29

The *#define* pre-processor directive **shall not** be used to create inline macros. Inline functions **shall** be used instead.

Rationale: Inline functions do not require text substitutions and behave well when called with arguments (e.g. type checking is performed). See AV Rule 29 in Appendix A for an example.

See section 4.13.6 for rules pertaining to inline functions.

AV Rule 30

The *#define* pre-processor directive **shall not** be used to define constant values. Instead, the *const* qualifier **shall** be applied to variable declarations to specify constant values.

Exception: The only exception to this rule is for constants that are commonly defined by third-party modules. For example, *#define* is typically used to define *NULL* in standard header files. Consequently, *NULL* may be treated as a macro for compatibility with third-party tools.

Rationale: *const* variables follow scope rules, are subject to type checking and do not require text substitutions (which can be confusing or misleading). See AV Rule 30 in Appendix A for an example.

AV Rule 31

The *#define* pre-processor directive **will** only be used as part of the technique to prevent multiple inclusions of the same header file.

Rationale: *#define* can be used to specify conditional compilation (AV Rule 27 and AV Rule 28), inline macros (AV Rule 29) and constants (AV Rule 30). This rule specifies that the only allowable use of *#define* is to prevent multiple includes of the same header file (AV Rule 27).

4.6.3 #include Pre-Processing Directive

AV Rule 32

The *#include* pre-processor directive **will** only be used to include header (*.h) files.

Exception: In the case of template class or function definitions, the code may be partitioned into separate header and implementation files. In this case, the implementation file may be included as a part of the header file. The implementation file is logically a part of the header and is not separately compilable. See AV Rule 32 in Appendix A.

Rationale: Clarity. The only files included in a .cpp file should be the relevant header (*.h) files.

4.7 Header Files

AV Rule 33

The *#include* directive **shall** use the <*filename.h*> notation to include header files.

Note that relative pathnames may also be used. See also AV Rule 53, AV Rule 53.1, and AV Rule 55 for additional information regarding header file names.

Rationale: The include form *"filename.h"* is typically used to include local header files. However, due to the unfortunate divergence in vendor implementations, only the <*filename.h*> form will be used.

Examples:

```
#include <foo.h>              // Good
#include <dir1/dir2/foo.h>    // Good: relative path  used
#include "foo.h"              // Bad: "filename.h" form used
```

AV Rule 34

Header files **should** contain logically related declarations only.

Rationale: Minimize unnecessary dependencies.

AV Rule 35

A header file **will** contain a mechanism that prevents multiple inclusions of itself.

Rationale: Avoid accidental header file recursion. Note AV Rule 27 specifies the mechanism by which multiple inclusions are to be eliminated whereas this rule (AV Rule 35) specifies that *each* header file *must* use that mechanism.

AV Rule 36

Compilation dependencies **should** be minimized when possible. (Stroustrup [2], Meyers [6], item 34)

Rationale: Minimize unnecessary recompilation of source files. See AV Rule 36 in Appendix A for an example.

Note: AV Rule 37 and AV Rule 38 detail several mechanisms by which compilation dependencies may be minimized.

AV Rule 37

Header (include) files **should** include only those header files that are required for them to successfully compile. Files that are only used by the associated .cpp file should be placed in the .cpp file—not the .h file.

Rationale: The *#include* statements in a header file define the dependencies of the file. Fewer dependencies imply looser couplings and hence a smaller *ripple-effect* when the header file is required to change.

AV Rule 38

Declarations of classes that are only accessed via pointers (*) or references (&) **should** be supplied by *forward headers* that contain only *forward declarations*.

Rationale: The header files of classes that are only referenced via pointers or references need not be included. Doing so often increases the coupling between classes, leading to increased compilation dependencies as well as greater maintenance efforts. *Forward declarations* of the classes in question (supplied by *forward headers*) can be used to limit implementation dependencies, maintenance efforts and compile times. See AV Rule 38 in Appendix A for an example. Note that this technique is employed in the standard header **<iosfwd>** to declare forward references to template classes used throughout <**iostreams**>.

AV Rule 39

Header files (*.h) **will not** contain non-const variable definitions or function definitions. (See also AV Rule 139.)

Rationale: Header files should typically contain interface declarations—not implementation details.

Exception: Inline functions and template definitions may be included in header files. See AV Rule 39 in Appendix A for an example.

4.8 Implementation Files

AV Rule 40

Every implementation file **shall** include the header files that uniquely define the inline functions, types, and templates used.

Rationale: Insures consistency checks. (See AV Rule 40 Appendix in A for additional details)

Note that this rule implies that the definition of a particular inline function, type, or template will never occur in multiple header files.

4.9 Style

Imposing constraints on the format of syntactic elements makes source code easier to read due to consistency in form and appearance. Note that automatic code generators should be configured to produce code that conforms to the style guidelines where possible. However, an exception is made for code generators that cannot be reasonably configured to comply with **should** or **will** style rules (safety-critical **shall** rules must still be followed).

AV Rule 41

Source lines **will** be kept to a length of 120 characters or less.

Rationale: Readability and style. Very long source lines can be difficult to read and understand.

AV Rule 42

Each expression-statement **will** be on a separate line.

Rationale: Simplicity, readability, and style. See AV Rule 42 in Appendix A for examples.

AV Rule 43

Tabs **should** be avoided.

Rationale: Tabs are interpreted differently across various editors and printers.

Note: many editors can be configured to map the 'tab' key to a specified number of spaces.

AV Rule 44

All indentations **will** be at least two spaces and be consistent within the same source file.

Rationale: Readability and style.

4.9.1 Naming Identifiers

The choice of identifier names should**:**

- Suggest the usage of the identifier.
- Consist of a descriptive name that is short yet meaningful.
- Be long enough to avoid name conflicts, but not excessive in length.
- Include abbreviations that are generally accepted.

Note: In general, the above guidelines should be followed. However, conventional usage of simple identifiers (*i, x, y, p,* etc.) in small scopes can lead to cleaner code and will therefore be permitted.

Additionally, the term 'word' in the following naming convention rules may be used to refer to a word, an acronym, an abbreviation, or a number.

AV Rule 45

All words in an identifier **will** be separated by the '_' character.

Rationale: Readability and Style.

AV Rule 46 (MISRA Rule 11, Revised)

User-specified identifiers (internal and external) **will not** rely on significance of more than 64 characters.

Note: The C++ standard suggests that a minimum of 1,024 characters will be significant. [10]

AV Rule 47

Identifiers **will not** begin with the underscore character '_'.

Rationale: '_' is often used as the first character in the name of library functions (e.g. *_main, _exit,* etc.) In order to avoid name collisions, identifiers should not begin with '_'.

AV Rule 48

Identifiers **will not** differ by:

- Only a mixture of case
- The presence/absence of the underscore character
- The interchange of the letter ‘O’, with the number ‘0’ or the letter ‘D’
- The interchange of the letter ‘I’, with the number ‘1’ or the letter ‘l’
- The interchange of the letter ‘S’ with the number ‘5’
- The interchange of the letter ‘Z’ with the number 2
- The interchange of the letter ‘n’ with the letter ‘h’.

Rationale: Readability.

AV Rule 49

All acronyms in an identifier **will** be composed of uppercase letters.

Note: An acronym will always be in upper case, even if the acronym is located in a portion of an identifier that is specified to be lower case by other rules.

Rationale: Readability.

4.9.1.1 Naming Classes, Structures, Enumerated types and typedefs

AV Rule 50

The first word of the name of a class, structure, namespace, enumeration, or type created with *typedef* **will** begin with an uppercase letter. All others letters **will** be lowercase.

Rationale: Style.

Example:

```
class Diagonal_matrix { … };          // Only first letter is capitalized;
enum RGB_colors {red, green, blue};   // RGB is an acronym so all letters are un upper case
```

Exception: The first letter of a *typedef* name may be in lowercase in order to conform to a standard library interface or when used as a replacement for fundamental types (see AV Rule 209).

```
typename C::value_type s=0;           // value_type of container C begins with a lower case
                                      //letter in conformance with standard library typedefs
```

4.9.1.2 Naming Functions, Variables and Parameters

AV Rule 51

All letters contained in function and variable names **will** be composed entirely of lowercase letters.

Rationale: Style.

Example:

```
class   Example_class_name
{
  public:
     uint16       example_function_name (void);
  private:
     uint16       example_variable_name;
};
```

4.9.1.3 Naming Constants and Enumerators

AV Rule 52

Identifiers for constant and enumerator values **shall** be lowercase.

Example:

```
const    uint16    max_pressure = 100;
enum     Switch_position          {up, down};
```

Rationale: Although it is an accepted convention to use uppercase letters for constants and enumerators, it is possible for third party libraries to replace constant/enumerator names as part of the macro substitution process (macros are also typically represented with uppercase letters).

4.9.2 Naming Files

Naming files should follow the same guidelines as naming identifiers with a few additions.

AV Rule 53

Header files **will** always have a file name extension of ".h".

AV Rule 53.1

The following character sequences **shall not** appear in header file names: ‘, \, /*, //, or ".

Rationale: If any of the character sequences ‘, \, /*, //, or " appears in a header file name (i.e. <h-char-sequence>), the resulting behavior is undefined. [10], 2.8(2) Note that relative pathnames may be used. However, only “/” may be used to separate directory and file names.

Examples:

```
#include <foo /* comment */ .h>        // Bad: “/*” prohibited
#include <foo’s .h>                    // Bad: “’” prohibited
#include <dir1\dir2\foo.h>             // Bad: “\” prohibited
#include <dir1/dir2/foo.h>             // Good: relative path  used
```

AV Rule 54

Implementation files **will** always have a file name extension of ".cpp".

AV Rule 55

The name of a header file **should** reflect the logical entity for which it provides declarations.

Example:

For the *Matrix* entity, the header file would be named:

Matrix.h

AV Rule 56

The name of an implementation file **should** reflect the logical entity for which it provides definitions and have a ".cpp" extension (this name will normally be identical to the header file that provides the corresponding declarations.)

At times, more than one .cpp file for a given logical entity will be required. In these cases, a suffix should be appended to reflect a logical differentiation.

Example 1: One .cpp file for the *Matrix* class:

Matrix.cpp

Example 2: Multiple files for a math library:

Math_sqrt.cpp
Math_sin.cpp
Math_cos.cpp

4.9.3 Classes

AV Rule 57

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

Rationale: By placing the public section first, everything that is of interest to a user is gathered in the beginning of the class definition. The protected section may be of interest to designers when considering inheriting from the class. The private section contains details that should be of the least general interest.

4.9.4 Functions

AV Rule 58

When declaring and defining functions with more than two parameters, the leading parenthesis and the first argument **will** be written on the same line as the function name. Each additional argument **will** be written on a separate line (with the closing parenthesis directly after the last argument).

Rationale: Readability and style. See AV Rule 58 in Appendix A for examples.

4.9.5 Blocks

AV Rule 59 (MISRA Rule 59, Revised)

The statements forming the body of an *if, else if, else, while, do…while* or *for* statement **shall** always be enclosed in braces, even if the braces form an empty block.

Rationale: Readability. It can be difficult to see ";" when it appears by itself. See AV Rule 59 in Appendix A for examples.

AV Rule 60

Braces ("{}") which enclose a block **will** be placed in the same column, on separate lines directly before and after the block.

Example:

```
if (var_name == true)
{
}
else
{
}
```

AV Rule 61

Braces ("{}") which enclose a block **will** have nothing else on the line except comments (if necessary).

4.9.6 Pointers and References

AV Rule 62

The dereference operator '*' and the address-of operator '&' **will** be directly connected with the type-specifier.

Rationale: The *int32* p;* form emphasizes type over syntax while the *int32 *p;* form emphasizes syntax over type. Although both forms are equally valid C++, the heavy emphasis on types in C++ suggests that *int32* p;* is the preferable form.

Examples:

```
int32*  p;      // Correct
int32  *p;      // Incorrect
int32*  p, q;   // Probably error. However, this declaration cannot occur
                // under the one name per declaration style required by AV Rule 152.
```

4.9.7 Miscellaneous

AV Rule 63

Spaces **will not** be used around '.' or '->', nor between unary operators and operands.

Rationale: Readability and style.

4.10 Classes

4.10.1 Class Interfaces

AV Rule 64

A class interface **should** be complete and minimal. See Meyers [6], item 18.

Rationale: A complete interface allows clients to do anything they may reasonably want to do. On the other hand, a minimal interface will contain as few functions as possible (i.e. no two functions will provide overlapping services). Hence, the interface will be no more complicated than it has to be while allowing clients to perform whatever activities are reasonable for them to expect.

Note: Overlapping services may be required where efficiency requirements dictate. Also, the use of helper functions (Stroustrup [2], 10.3.2) can simplify class interfaces.

4.10.2 Considerations Regarding Access Rights

Roughly two types of classes exist: those that essentially aggregate data and those that provide an abstraction while maintaining a well-defined state or invariant. The following rules provide guidance in this regard.

AV Rule 65

A structure **should** be used to model an entity that does not require an invariant.

AV Rule 66

A class **should** be used to model an entity that maintains an invariant.

AV Rule 67

Public and protected data **should** only be used in structs—not classes.

Rationale: A class is able to maintain its invariant by controlling access to its data. However, a class cannot control access to its members if those members non-private. Hence all data in a class should be private.

Exception: Protected members may be used in a class as long as that class does not participate in a client interface. See AV Rule 88.

4.10.3 Member Functions

AV Rule 68

Unneeded implicitly generated member functions **shall** be explicitly disallowed. See Meyers [6], item 27.

Rationale: Eliminate any surprises that may occur as a result of compiler generated functions. For example, if the assignment operator is unneeded for a particular class, then it should be declared *private* (and not defined). Any attempt to invoke the operator will result in a compile-time error. On the contrary, if the assignment operator is not declared, then when it is invoked, a compiler-generated form will be created and subsequently executed. This could lead to unexpected results.

Note: If the copy constructor is explicitly disallowed, the assignment operator should be as well.)

4.10.4 const Member Functions

AV Rule 69

A member function that does not affect the state of an object (its instance variables) **will** be declared *const*.

Member functions should be *const* by default. Only when there is a clear, explicit reason should the *const* modifier on member functions be omitted.

Rationale: Declaring a member function *const* is a means of ensuring that objects will not be modified when they should not. Furthermore, C++ allows member functions to be overloaded on their *const*-ness.

4.10.5 Friends

AV Rule 70

A class **will** have friends only when a function or object requires access to the private elements of the class, but is unable to be a member of the class for logical or efficiency reasons.

Rationale: The overuse of friends leads to code that is both difficult to understand and maintain.

AV Rule 70 in Appendix A provides examples of acceptable uses of friends. Note that the alternative to friendship in some instances is to expose more internal detail than is necessary. In those cases friendship is not only allowed, but is the preferable option.

4.10.6 Object Lifetime, Constructors, and Destructors

4.10.6.1 Object Lifetime

Conceptually, developers understand that objects should not be used before they have been created or after they have been destroyed. However, a number of scenarios may arise where this distinction may not be obvious. Consequently, the following object-lifetime rule is provided to highlight these instances.

AV Rule 70.1

An object **shall not** be improperly used before its lifetime begins or after its lifetime ends.

Rationale: Improper use of an object, before it is created or after it is destroyed, results in undefined behavior. See section 3.8 of [10] for details on “proper” vs. “improper” use. See also AV Rule 70.1 in Appendix A for examples.

4.10.6.2 Constructors

AV Rule 71

Calls to an externally visible operation of an object, other than its constructors, **shall not** be allowed until the object has been fully initialized.

Rationale: Avoid problems resulting from incomplete object initialization. Further details are given in AV Rule 71 in Appendix A.

AV Rule 71.1

A class's virtual functions **shall not** be invoked from its destructor or any of its constructors.

Rationale: A class's virtual functions are resolved statically (not dynamically) in its constructors and destructor. See AV Rule 71.1 in Appendix A for additional details.

AV Rule 72

The invariant[2] for a class **should** be:

- a part of the postcondition of every class constructor,
- a part of the precondition of the class destructor (if any),
- a part of the precondition and postcondition of every other publicly accessible operation.

Rationale: Prohibit clients from influencing the invariant of an object through any other means than the public interface.

AV Rule 73

Unnecessary default constructors **shall not** be defined. See Meyers [7], item 4. (See also AV Rule 143).

Rationale: Discourage programmers from creating objects until the requisite data is available for complete object construction (i.e. prevent objects from being created in a *partially initialized* state). See AV Rule 73 in Appendix A for examples.

AV Rule 74

Initialization of nonstatic class members **will** be performed through the member initialization list rather than through assignment in the body of a constructor. See Meyers [6], item 12.

Exception: Assignment should be used when an initial value cannot be represented by a simple expression (e.g. initialization of array values), or when a name must be introduced before it can be initialized (e.g. value received via an input stream).

See AV Rule 74 in Appendix A for details.

[2] A class invariant is a statement-of-fact about a class that must be true for all *stable* instances of the class. A class is considered to be in a *stable state* immediately after construction, immediately before destruction, and immediately before and after any remote public method invocation.

AV Rule 75

Members of the initialization list **shall** be listed in the order in which they are declared in the class. See Stroustrup [2], 10.4.5 and Meyers [6], item 13.

Note: Since base class members are initialized before derived class members, base class initializers should appear at the beginning of the member initialization list.

Rationale: Members of a class are initialized in the order in which they are declared—not the order in which they appear in the initialization list.

AV Rule 76

A copy constructor and an assignment operator **shall** be declared for classes that contain pointers to data items or nontrivial destructors. See Meyers [6], item 11.

Note: See also AV Rule 80 which indicates that default copy and assignment operators are preferable when those operators offer reasonable semantics.

Rationale: Ensure resources are appropriately managed during copy and assignment operations. See AV Rule 76 in Appendix A for additional details.

AV Rule 77

A copy constructor **shall** copy all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).

Note: If a reference counting mechanism is employed by a class, a literal copy need not be performed in every case. See also AV Rule 83.

Rationale: Ensure data members and bases are properly handled when an object is copied. See AV Rule 77 in Appendix A for additional details.

AV Rule 77.1

The definition of a member function **shall not** contain default arguments that produce a signature identical to that of the implicitly-declared copy constructor for the corresponding class/structure.

Rationale: Compilers are not required to diagnose this ambiguity. See AV Rule 77.1 in Appendix A for additional details.

4.10.6.3 Destructors

AV Rule 78

All base classes with a virtual function **shall** define a virtual destructor.

Rationale: Prevent undefined behavior. If an application attempts to delete a derived class object through a base class pointer, the result is undefined if the base class's destructor is non-virtual.

Note: This rule does not imply the use of dynamic memory (allocation/deallocation from the free store) will be used. See AV Rule 206.

AV Rule 79

All resources acquired by a class **shall** be released by the class's destructor. See Stroustrup [2], 14.4 and Meyers [7], item 9.

Rationale: Prevention of resource leaks, especially in error cases. See AV Rule 79 in Appendix A for additional details.

4.10.7 Assignment Operators

AV Rule 80

The default copy and assignment operators **will** be used for classes when those operators offer reasonable semantics.

Rationale: The default versions are more likely to be correct, easier to maintain and efficient than that generated by hand.

AV Rule 81

The assignment operator **shall** handle self-assignment correctly (see Stroustrup [2], Appendix E.3.3 and 10.4.4)

Rationale: *a = a;* must function correctly. See AV Rule 81 in Appendix A for examples.

AV Rule 82

An assignment operator **shall** return a reference to **this*.

Rationale: Both the standard library types and the built-in types behave in this manner. See AV Rule 81 for an example of an assignment operator overload.

AV Rule 83

An assignment operator **shall** assign all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).

Note: To correctly copy a stateful virtual base in a portable manner, it must hold that if *x1* and *x2* are objects of virtual base *X*, then *x1=x2; x1=x2;* must be semantically equivalent to *x1=x2;* [10] 12.8(13)

Rationale: Ensure data members and bases are properly handled under assignment. See AV Rule 83 in Appendix A for additional details. See also AV Rule 77.

4.10.8 Operator Overloading

AV Rule 84

Operator overloading **will** be used sparingly and in a conventional manner.

Rationale: Since unconventional or inconsistent uses of operator overloading can easily lead to confusion, operator overloads should only be used to enhance clarity and should follow the natural meanings and conventions of the language. For instance, a C++ operator "+=" shall have the same meaning as "+" and "=".

AV Rule 85

When two operators are opposites (such as == and !=), both **will** be defined and one **will** be defined in terms of the other.

Rationale: If *operator==()* is supplied, then one could reasonable expect that *operator!=()* would be supplied as well. Furthermore, defining one in terms of the other simplifies maintenance. See AV Rule 85 in Appendix A for an example.

4.10.9 Inheritance

Class hierarchies are appropriate when run-time selection of implementation is required. If run-time resolution is not required, template parameterization should be considered (templates are better-behaved and faster than virtual functions). Finally, simple independent concepts should be expressed as concrete types. The method selected to express the solution should be commensurate with the complexity of the problem.

The following rules provide additional detail and guidance when considering the structure of inheritance hierarchies.

AV Rule 86

Concrete types **should** be used to represent simple independent concepts. See Stroustrup [2], 25.2.

Rationale: Well designed concrete classes tend to be efficient in both space and time, have minimal dependencies on other classes, and tend to be both comprehensible and usable in isolation.

AV Rule 87

Hierarchies **should** be based on abstract classes. See Stroustrup [2], 12.5.

Rationale: Hierarchies based on abstract classes tend to focus designs toward producing clean interfaces, keep implementation details out of interfaces, and minimize compilation dependencies while allowing alternative implementations to coexist. See AV Rule 87 in Appendix A for examples.

AV Rule 88

Multiple inheritance **shall** only be allowed in the following restricted form: *n* interfaces plus *m* private implementations, plus at most one protected implementation.

Rationale: Multiple inheritance can lead to complicated inheritance hierarchies that are difficult to comprehend and maintain.

See AV Rule 88 in Appendix A for examples of both appropriate and inappropriate uses of multiple inheritance.

AV Rule 88.1

A stateful virtual base shall be explicitly declared in each derived class that accesses it.

Rationale: Explicitly declaring a stateful virtual base at each level in a hierarchy (where that base is used), documents that fact that no assumptions can be made with respect to the exclusive use of the data contained within the virtual base. See AV Rule 88.1 in Appendix A for additional details.

AV Rule 89

A base class **shall not** be both virtual and non-virtual in the same hierarchy.

Rationale: Hierarchy becomes difficult to comprehend and use.

AV Rule 90

Heavily used interfaces **should** be minimal, general and abstract. See Stroustrup [2] 23.4.

Rationale: Enable interfaces to exhibit stability in the face of changes to their hierarchies.

AV Rule 91

Public inheritance **will** be used to implement "is-a" relationships. See Meyers [6], item 35.

Rationale: Public inheritance and private inheritance mean very different things in C++ and should therefore be used accordingly. Public inheritance implies an "is-a" relationship. That is, every object of a publicly derived class *D* is also an object of the base type *B*, but not vice versa. Moreover, type *B* represents a more general concept than type *D*, and type *D* represents a more specialized concept than type *B*. Thus, stating that *D* publicly inherits from *B*, is an assertion that *D* is a subtype of *B*. That is, objects of type *D* may be used anywhere that objects of type *B* may be used (since an object of type *D* is really an object of type *B* as well).

In contrast to public inheritance, private and protected inheritance means "is-implemented-in-terms-of". It is purely an implementation technique—the interface is ignored. See also AV Rule 93.

AV Rule 92

A subtype (publicly derived classes) **will** conform to the following guidelines with respect to all classes involved in the polymorphic assignment of different subclass instances to the same variable or parameter during the execution of the system:

- Preconditions of derived methods must be at least as weak as the preconditions of the methods they override.
- Postconditions of derived methods must be at least as strong as the postconditions of the methods they override.

In other words, subclass methods must expect less and deliver more than the base class methods they override. This rule implies that subtypes will conform to the Liskov Substitution Principle.

Rationale: Predictable behavior of derived classes when used within base class contexts. See AV Rule 92 in Appendix A for additional details.

AV Rule 93

"has-a" or "is-implemented-in-terms-of" relationships **will** be modeled through membership or non-public inheritance. See Meyers [6], item 40.

Rationale: Public inheritance means "is-a" (see AV Rule 91) while nonpublic inheritance means "has-a" or "is-implemented-in-terms-of". See AV Rule 93 in Appendix A for examples.

AV Rule 94

An inherited nonvirtual function **shall not** be redefined in a derived class. See Meyers [6], item 37.

Rationale: Prevent an object from exhibiting "two-faced" behavior. See AV Rule 94 in Appendix A for an example.

AV Rule 95

An inherited default parameter **shall never** be redefined. See Meyers [6], item 38.

Rationale: The redefinition of default parameters for virtual functions often produces surprising results. See AV Rule 95 in Appendix A for an example.

AV Rule 96

Arrays **shall not** be treated polymorphically. See Meyers [7], item 3.

Rationale: Array indexing in C/C++ is implemented as pointer arithmetic. Hence, a[i] is equivalent to a+i*SIZEOF(array element). Since derived classes are often larger than base classes, polymorphism and pointer arithmetic are not compatible techniques.

AV Rule 97

Arrays **shall not** be used in interfaces. Instead, the *Array* class should be used.

Rationale: Arrays degenerate to pointers when passed as parameters. This “array decay” problem has long been known to be a source of errors.

Note: See Array.doc for guidance concerning the proper use of the *Array* class, including its interaction with memory management and error handling facilities.

4.10.10Virtual Member Functions

AV Rule 97.1

Neither operand of an equality operator (== or !=) **shall** be a pointer to a virtual member function.

Rationale: If either operand of an equality operator (== or !=) is a pointer to a virtual member function, the result is unspecified [10], 5.10(2).

Several other sections have also touched on virtual member functions and polymorphism. Hence, the following cross references are provided so that these rules may be accessed from a single location: AV Rule 71, AV Rule 78, AV Rule 87-AV Rule 97, and AV Rule 221.

4.11 Namespaces

AV Rule 98

Every nonlocal name, except main(), **should** be placed in some namespace. See Stroustrup [2], 8.2.

Rationale: Avoid name clashes in large programs with many parts.

AV Rule 99

Namespaces **will not** be nested more than two levels deep.

Rationale: Simplicity and clarity. Deeply nested namespaces can be difficult to comprehend and use correctly.

AV Rule 100

Elements from a namespace **should** be selected as follows:

- *using declaration* or *explicit qualification* for few (approximately five) names,
- *using directive* for many names.

Rationale: All elements in a namespace need not be pulled in if only a few elements are actually needed.

4.12 Templates

Templates provide a powerful technique for creating families of functions or classes parameterized by *type*. As a result, *generic components* may be created that match corresponding *hand-written* versions in both size and performance [2].

Although template techniques have proven to be both powerful and expressive, it may be unclear when to prefer the use of templates over the use of inheritance. The following guidelines provided by Stroustrup[2], 13.8, offer advice in this regard:

1. Prefer a template over derived classes when run-time efficiency is at a premium.
2. Prefer derived classes over a template if adding new variants without recompilation is important.
3. Prefer a template over derived classes when no common base can be defined.
4. Prefer a template over derived classes when built-in types and structures with compatibility constraints are important.

AV Rule 101

Templates **shall** be reviewed as follows:

1. with respect to the template in isolation considering assumptions or requirements placed on its arguments.
2. with respect to all functions instantiated by actual arguments.

Note: The compiler should be configured to generate the list of actual template instantiations. See AV Rule 101 in Appendix A for an example.

Rationale: Since many instantiations of a template can be generated, any review should consider all actual instantiations as well as any assumptions or requirements placed on arguments of instantiations.

AV Rule 102

Template tests **shall** be created to cover all actual template instantiations.

Note: The compiler should be configured to generate the list of actual template instantiations. See AV Rule 102 in Appendix A for an example.

Rationale: Since many instantiations of a template can be generated, test cases should be created to cover all actual instantiations.

AV Rule 103

Constraint checks **should** be applied to template arguments.

Rationale: Explicitly capture parameter constraints in code as well as produce comprehensible error messages. See AV Rule 103 in Appendix A for examples.

AV Rule 104

A template specialization **shall be** declared before its use. See Stroustrup [2], 13.5.

Rationale: C++ language rule. The specialization must be in scope for every use of the type for which it is specialized.

Example:

```
template<class T> class List {…};
List<int32*> li;
template<class T> class List<T*> {…};        //Error: this specialization should be used for li
                                             // in the previous statement.
```

AV Rule 105

A template definition's dependence on its instantiation contexts **should** be minimized. See Stroustrup [2], 13.2.5 and C.13.8.

Rationale: Since templates are likely to be instantiated in multiple contexts with different parameter types, any nonlocal dependencies will increase the likelihood that errors or incompatibilities will eventually surface.

AV Rule 106

Specializations for pointer types **should** be made where appropriate. See Stroustrup [2], 13.5.

Rationale: Pointer types often require special semantics or provide special optimization opportunities.

4.13 Functions

4.13.1 Function Declaration, Definition and Arguments

AV Rule 107 (MISRA Rule 68)

Functions **shall** always be declared at file scope.

Rationale: Declaring functions at block scope may be confusing.

AV Rule 108 (MISRA Rule 69)

Functions with variable numbers of arguments **shall not** be used.

Rationale: The *variable argument* feature is difficult to use in a type-safe manner (i.e. typical language checking rules aren't applied to the additional parameters).

Note: In some cases, default arguments and overloading are alternatives to the *variable arguments* feature. See AV Rule 108 in Appendix A for an example.

AV Rule 109

A function definition **should not** be placed in a class specification unless the function is intended to be inlined.

Rationale: Class specifications are less compact and more difficult to read when they include implementations of member functions. Consequently, it is often preferable to place member function implementations in a separate file. However, including the implementation in the specification instructs the compiler to inline the method (if possible). Since inlining *short* functions can save both time and space, functions intended to be inlined may appear in the class specification. See AV Rule 109 in Appendix A for an example.

AV Rule 110

Functions with more than 7 arguments **will** not be used.

Rationale: Functions having long argument lists can be difficult to read, use, and maintain. Functions with too many parameters may indicate an under use of objects and abstractions.
Exception: Some constructors may require more than 7 arguments. However, one should consider if abstractions are being underused in this scenario.

AV Rule 111

A function **shall not** return a pointer or reference to a non-static local object.

Rationale: After return, the local object will no longer exist.

AV Rule 112

Function return values **should not** obscure resource ownership.

Rationale: Potential source of resource leaks. See AV Rule 173 and AV Rule 112 in Appendix A for examples.

4.13.2 Return Types and Values

AV Rule 113 (MISRA Rule 82, Revised)

Functions **will** have a single exit point.

Rationale: Numerous exit points tend to produce functions that are both difficult to understand and analyze.

Exception: A single exit is not required if such a structure would obscure or otherwise significantly complicate (such as the introduction of additional variables) a function's control logic. Note that the usual resource clean-up must be managed at all exit points.

AV Rule 114 (MISRA Rule 83, Revised)

All exit points of value-returning functions **shall** be through return statements.

Rationale: Flowing off the end of a value-returning function results in undefined behavior.

AV Rule 115 (MISRA Rule 86)

If a function returns error information, then that error information **will** be tested.

Rationale: Ignoring return values could lead to a situation in which an application continues processing under the false assumption that the context in which it is operating (or the item on which it is operating) is valid.

4.13.3 Function Parameters (Value, Pointer or Reference)

AV Rule 116

Small, concrete-type arguments (two or three words in size) **should** be passed by value if changes made to formal parameters should not be reflected in the calling function.

Rationale: *Pass-by-value* is the simplest, safest method for small objects of concrete type. Note that non-concrete objects must be passed by pointer or reference to realize polymorphic behavior. See rules AV Rule 117 and AV Rule 118.

AV Rule 117

Arguments **should** be passed by reference if NULL values are not possible:

AV Rule 117.1 An object should be passed as *const T&* if the function should not change the value of the object.

AV Rule 117.2 An object should be passed as *T&* if the function may change the value of the object.

Rationale: Since references cannot be NULL, checks for NULL values will be eliminated from the code. Furthermore, references offer a more convenient notation than pointers.

AV Rule 118

Arguments **should** be passed via pointers if NULL values are possible:

AV Rule 118.1 An object should be passed as *const T** if its value should not be modified.

AV Rule 118.2 An object should be passed as *T** if its value may be modified.

Rationale: References cannot be NULL.

4.13.4 Function Invocation

AV Rule 119 (MISRA Rule 70)

Functions **shall not** call themselves, either directly or indirectly (i.e. recursion **shall not** be allowed).

Rationale: Since stack space is not unlimited, stack overflows are possible.

Exception: Recursion will be permitted under the following circumstances:

1. development of SEAL 3 or general purpose software, or
2. it can be proven that adequate resources exist to support the maximum level of recursion possible.

4.13.5 Function Overloading

AV Rule 120

Overloaded operations or methods **should** form families that use the same semantics, share the same name, have the same purpose, and that are differentiated by formal parameters.

Rationale: Inconsistent use of overloading can lead to considerable confusion. See AV Rule 120 in Appendix A for examples.

4.13.6 Inline Functions

Inline functions often offer a speed advantage over traditional functions as they do not require the typical function call overhead. Functions are typically inlined when either the function definition is included in the class declaration or the keyword *inline* precedes the function definition.

Example A: Inlined since definition is included in declaration.

```
class Sample_class
{
 public:
      int32  get_data (void)
     {
       return data;
     }
};
```

Example B: Inlined because of the *inline* keyword

```
int32 foo (void);

inline int foo (void)
{
  …
}
```

The C++ standard [10] provides the following information in regards to the use of inline functions. These observations are not listed as **AV Rules** since they are C++ language rules.

1. An inline function shall be defined in every translation unit in which it is used and shall have exactly the same definition in every case. (Note this observation implies that inline function definitions should be included in header files.)
2. If a function with external linkage is declared inline in one translation unit, it shall be declared inline in all translation units in which it appears; no diagnostic is required.
3. An inline function with external linkage shall have the same address in all translation units.
4. A static local variable in an extern inline function always refers to the same object.
5. A string literal in an extern inline function is the same object in different translation units.

AV Rule 121

Only functions with 1 or 2 statements **should** be considered candidates for inline functions.

Rationale: The compiler is not compelled to actually make a function inline. Decision criteria differ from one compiler to another. The keyword *inline* is simply a request for the compiler to inline the function. The compiler is free to ignore this request and make a real function call. See AV Rule 121 in Appendix A for additional details.

AV Rule 122

Trivial accessor and mutator functions **should** be inlined.

Rationale: Inlining short, simple functions can save both time and space. See AV Rule 122 in Appendix A for an example.

AV Rule 123

The number of accessor and mutator functions **should** be minimized.

Rationale: Numerous accessors and mutators may indicate that a class simply serves to aggregate a collection of data rather than to embody an abstraction with a well-defined state or invariant. In this case, a struct with public data may be a better alternative (see section 4.10.2, AV Rule 65, and AV Rule 66).

AV Rule 124

Trivial forwarding functions **should** be inlined.

Rationale: Inlining short, simple functions can save both time and space.

4.13.7 Temporary Objects

AV Rule 125

Unnecessary temporary objects **should** be avoided. See Meyers [7], item 19, 20, 21.

Rationale: Since the creation and destruction of temporary objects that are either large or involve complicated constructions can result in significant performance penalties, they should be avoided. See AV Rule 125 in Appendix A for additional details.

4.14 Comments

Comments in header files are meant for the users of classes and functions, while comments in implementation files are meant for those who maintain the classes.

Comments are often said to be either strategic or tactical. A strategic comment describes what a function or section of code is intended to do, and is placed before the code. A tactical comment describes what a single line of code is intended to do. Unfortunately, too many tactical comments can make code unreadable. For this reason, comments should be primarily strategic, unless trying to explain very complicated code (i.e. one should avoid stating in a comment what is clearly stated in code).

AV Rule 126

Only valid C++ style comments (//) **shall** be used. See AV Rule 126 in Appendix A for additional details concerning *valid* C++ style comments.

Rationale: A single standard provides consistency throughout the code.

Exception: Automatic code generators that cannot be configured to use the "//" form.

AV Rule 127

Code that is not used (commented out) **shall** be deleted.

Rationale: No dead code is allowed.

Exception: Code that is simply part of an explanation may appear in comments.

AV Rule 128

Comments that document actions or sources (e.g. tables, figures, paragraphs, etc.) outside of the file being documented **will** not be allowed.

Rationale: The comments in a file should require changes only when changes are necessary to the file itself. Note that this rule does not preclude the documentation of valid assumptions that may made be entities contained within the file.

AV Rule 129

Comments in header files **should** describe the externally visible behavior of the functions or classes being documented.

Rationale: Exposing the internal workings of functions or classes to clients might enable those clients to create dependences on the internal representations.

AV Rule 130

The purpose of every line of executable code **should** be explained by a comment, although one comment may describe more than one line of code.

Rationale: Readability. Every line of code should be represented by a comment. However, this rule does not say that every line of code should have a comment; a comment might represent more than one source line of code.

AV Rule 131

One **should** avoid stating in comments what is better stated in code (i.e. do not simply repeat what is in the code).

Rationale: While redundant comments are unnecessary, they also serve to increase the maintenance effort.

Example: The following example illustrates an unnecessary comment.

```
a = b+c;          // Bad: add b to c and place the result in a.
```

AV Rule 132

Each variable declaration, typedef, enumeration value, and structure member **will** be commented.

Exception: Cases where commenting would be unnecessarily redundant.

AV Rule 133

Every source file **will** be documented with an introductory comment that provides information on the file name, its contents, and any program-required information (e.g. legal statements, copyright information, etc).

AV Rule 134

Assumptions (limitations) made by functions **should** be documented in the function's preamble.

Rationale: Maintenance efforts become very difficult if the assumptions (limitations) upon which functions are built are unknown.

4.15 Declarations and Definitions

AV Rule 135 (MISRA Rule 21, Revised)

Identifiers in an inner scope **shall not** use the same name as an identifier in an outer scope, and therefore hide that identifier.

Rationale: Hiding identifiers can be very confusing.

Example:

```
int32    sum = 0;
{
    int32        sum = 0;        // Bad: hides sum in outer scope.
    …
    sum = f (x);
}
```

AV Rule 136 (MISRA Rule 22, Revised)

Declarations **should** be at the smallest feasible scope. (See also AV Rule 143).

Rationale: This rule attempts to minimize the number of *live variables* that must be simultaneously considered. Furthermore, variable declarations should be postponed until enough information is available for full initialization (i.e. a variable should never be placed in a *partly-initialized* or *initialized-but-not-valid* state.) See AV Rule 136 in Appendix A for examples.

AV Rule 137 (MISRA Rule 23)

All declarations at file scope **should** be static where possible.

Rationale: Minimize dependencies between translation units where possible. See AV Rule 137 in Appendix A for additional details.

AV Rule 138 (MISRA Rule 24)

Identifiers **shall not** simultaneously have both internal and external linkage in the same translation unit.

Rationale: Avoid variable-name hiding which can be confusing. See AV Rule 138 in Appendix A for further details.

AV Rule 139 (MISRA Rule 27)

External objects **will** not be declared in more than one file. (See also AV Rule 39.)

Rationale: Avoid inconsistent declarations. See AV Rule 139 in Appendix A for further details.

Note: This type of error will be caught by linkers, but typically later than is desired (i.e. the inconsistency could exist in a different group's build.) Normally this will mean declaring external objects in header files which will then be included in all other files that need to use those objects (including the files which define them).

AV Rule 140 (MISRA Rule 28, Revised)

The *register* storage class specifier **shall not** be used.

Rationale: Compiler technology is now capable of optimal register placement.

AV Rule 141

A class, structure, or enumeration **will** not be declared in the definition of its type.

Rationale: Readability. See AV Rule 141 in Appendix A for examples.

4.16 Initialization

AV Rule 142 (MISRA Rule 30, Revised)

All variables **shall** be initialized before use. (See also AV Rule 136, AV Rule 71, and AV Rule 73, and AV Rule 143 concerning declaration scope, object construction, default constructors, and the point of variable introduction respectively.)

Rationale: Prevent the use of variables before they have been properly initialized. See AV Rule 142 in Appendix A for additional information.

Exception: Exceptions are allowed where a name must be introduced before it can be initialized (e.g. value received via an input stream).

AV Rule 143

Variables **will not** be introduced until they can be initialized with meaningful values. (See also AV Rule 136, AV Rule 142, and AV Rule 73 concerning declaration scope, initialization before use, and default constructors respectively.)

Rationale: Prevent clients from accessing variables without meaningful values. See AV Rule 143 in Appendix A for examples.

AV Rule 144 (MISRA Rule 31)

Braces **shall** be used to indicate and match the structure in the non-zero initialization of arrays and structures.

Rationale: Readability.

Example:

```
int32    a[2][2] = { {0,1} ,{2,3} };
```

AV Rule 145 (MISRA Rule 32)

In an enumerator list, the '=' construct **shall not** be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Rationale: Mixing the automatic and manual allocation of enumerator values is error-prone. Note that exceptions are allowed for clearly-defined standard conventions. See AV Rule 145 in Appendix A for additional details.

4.17 Types

AV Rule 146 (MISRA Rule 15)

Floating point implementations **shall** comply with a defined floating point standard.

The standard that will be used is the ANSI/IEEE Std 754 [1].

Rationale: Consistency.

AV Rule 147 (MISRA Rule 16)

The underlying bit representations of floating point numbers **shall not** be used in any way by the programmer.

Rationale: Manipulating bits is error prone. See AV Rule 147 in Appendix A for additional details.

AV Rule 148

Enumeration types **shall** be used instead of integer types (and constants) to select from a limited series of choices.

Note: This rule is not intended to exclude character constants (e.g. 'A', 'B', 'C', etc.) from use as case labels.

Rationale: Enhances debugging, readability and maintenance. Note that a compiler flag (if available) should be set to generate a warning if all enumerators are not present in a switch statement.

4.18 Constants

Section 4.6.2 contains additional details concerning constants and the use of *enum* and *#define*.

AV Rule 149 (MISRA Rule 19)

Octal constants (other than zero) **shall not** be used.

Rationale: Any integer constant beginning with a zero ('0') is defined by the C++ standard to be an octal constant. Due to the confusion this causes, octal constants should be avoided.

Note: Hexadecimal numbers and zero (which is also an octal constant) are allowed.

AV Rule 150

Hexadecimal constants **will** be represented using all uppercase letters.

AV Rule 151

Numeric values in code **will not** be used; symbolic values **will** be used instead.

Rationale: Improved readability and maintenance.

Exception: A class/structure constructor may initialize an array member with numeric values.

```
class A
{
  A()
  {
    coefficient[0] = 1.23;  // Good
    coefficient[1] = 2.34;  // Good
    coefficient[2] = 3.45;  // Good
  }
  private:
    float64 coefficient[3];  // Cannot be initialized via the member initialization list.
};
```

Note: In many cases '0' and '1' are not *magic numbers* but are part of the fundamental logic of the code (e.g. '0' often represents a NULL pointer). In such cases, '0' and '1' may be used.

AV Rule 151.1

A string literal **shall not** be modified.

Note that strictly conforming compilers should catch violations, but many do not.

Rationale: The effect of attempting to modify a string literal is undefined [10], 2.13.4(2). See also AV Rule 151.1 in Appendix A for additional details.

4.19 Variables

AV Rule 152

Multiple variable declarations **shall not** be allowed on the same line.

Rationale: Increases readability and prevents confusion (see also AV Rule 62).

Example:

```
int32*    p, q;                                  // Probably error.
int32     first button_on_top_of_the_left_box, i;  // Bad: Easy to overlook i
```

4.20 Unions and Bit Fields

AV Rule 153 (MISRA Rule 110, Revised)

Unions **shall not** be used.

Rationale: Unions are not statically type-safe and are historically known to be a source of errors.

Note: In some cases, derived classes and virtual functions may be used as an alternative to unions.

AV Rule 154 (MISRA Rules 111 and 112, Revised)

Bit-fields **shall** have explicitly unsigned integral or enumeration types only.

Rationale: Whether a plain (neither explicitly signed nor unsigned) char, short, int or long bit-field is signed or unsigned is implementation-defined.[10] Thus, explicitly declaring a bit-filed unsigned prevents unexpected sign extension or overflow.

Note: MISRA Rule 112 no longer applies since it discusses a two-bit minimum-length requirement for bit-fields of signed *types*.

AV Rule 155

Bit-fields **will not** be used to pack data into a word for the sole purpose of saving space.

Note: Bit-packing should be reserved for use in interfacing to hardware or conformance to communication protocols.

Warning: Certain aspects of bit-field manipulation are implementation-defined.

Rationale: Bit-packing adds additional complexity to the source code. Moreover, bit-packing may not save any space at all since the reduction in data size achieved through packing is often offset by the increase in the number of instructions required to pack/unpack the data.

AV Rule 156 (MISRA Rule 113)

All the members of a structure (or class) **shall** be named and shall only be accessed via their names.

Rationale: Reading/writing to unnamed locations in memory is error prone.

Exception: An unnamed bit-field of width zero may be used to specify alignment of the next bit-field at an allocation boundary. [10], 9.6(2)

4.21 Operators

AV Rule 157 (MISRA Rule 33)

The right hand operand of a && or || operator **shall not** contain side effects.

Rationale: Readability. The conditional evaluation of the right-hand side could be overlooked. See AV Rule 157 in Appendix A for an example.

AV Rule 158 (MISRA Rule 34)

The operands of a logical && or || **shall** be parenthesized if the operands contain binary operators.

Rationale: Readability. See AV Rule 158 in Appendix A for examples.

AV Rule 159

Operators ||, &&, and unary & **shall not** be overloaded. See Meyers [7], item 7.

Rationale: First, the behavior of the || and && operators depend on short-circuit evaluation of the operands. However, short-circuit evaluation is not possible for overloaded versions of the || and && operators. Hence, overloading these operators may produce unexpected results.

Next, if the address of an object of incomplete class type is taken, but the complete form of the type declares operator&() as a member function, the resulting behavior is undefined. [10]

AV Rule 160 (MISRA Rule 35, Modified)

An assignment expression **shall** be used only as the expression in an expression statement.

Rationale: Readability. Assignment (=) may be easily confused with the equality (==). See AV Rule 160 in Appendix A for examples.

AV Rule 162

Signed and unsigned values **shall not** be mixed in arithmetic or comparison operations.

Rationale: Mixing signed and unsigned values is error prone as it subjects operations to numerous arithmetic conversion and integral promotion rules.

AV Rule 163

Unsigned arithmetic **shall not** be used.

Rationale: Over time, unsigned values will likely be mixed with signed values thus violating AV Rule 162.

AV Rule 164 (MISRA Rule 38)

The right hand operand of a shift operator **shall** lie between zero and one less than the width in bits of the left-hand operand (inclusive).

Rationale: If the right operand is either negative or greater than or equal to the length in bits of the promoted left operand, the result is undefined. [10]

AV Rule 164.1

The left-hand operand of a right-shift operator **shall not** have a negative value.

Rationale: For $e_1 >> e_2$, if e_1 has a signed type and a negative value, the value of ($e_1 >> e_2$) is implementation-defined. [10]

AV Rule 165 (MISRA Rule 39)

The unary minus operator **shall not** be applied to an unsigned expression.

AV Rule 166 (MISRA Rule 40)

The *sizeof* operator **will not** be used on expressions that contain side effects.

Rationale: Clarity. The side-effect will not be realized since *sizeof* only operates on the type of an expression: the expression itself will not be evaluated.

AV Rule 167 (MISRA Rule 41)

The implementation of integer division in the chosen compiler **shall** be determined, documented and taken into account.

Rationale: If one or more of the operands of an integer division is negative, the sign of the remainder is implementation defined. [10]

Note: For the Green Hills PowerPC C++ compiler, the sign of the remainder is the same as that of the first operand. Also the quotient is rounded toward zero.

AV Rule 168 (MISRA Rule 42, Revised)

The comma operator **shall not** be used.

Rationale: Readability. See AV Rule 168 in Appendix A for additional details.

4.22 Pointers & References

AV Rule 169

Pointers to pointers **should** be avoided when possible.

Rationale: Pointers to pointers are a source of bugs and result in obscure code. Containers or some other form of abstraction should be used instead (see AV Rule 97).

AV Rule 170 (MISRA Rule 102, Revised)

More than 2 levels of pointer indirection **shall not** be used.

Rationale: Multiple levels of pointer indirections typically produce code that is difficult to read, understand and maintain.

Note: This rule leaves no room for using more than 2 levels of pointer indirection. The word “shall” replaces the word “should” in MISRA Rule 102.

AV Rule 171 (MISRA Rule 103)

Relational operators **shall not** be applied to pointer types except where both operands are of the same type and point to:

- the same object,
- the same function,
- members of the same object, or
- elements of the same array (including one past the end of the same array).

Note that if either operand is null, then both **shall** be null. Also, "members of the same object" should not be construed to include base class subobjects (See also AV Rule 210).

Rationale: Violations of the above rule may result in unspecified behavior [10], 5.9(2).

AV Rule 173 (MISRA Rule 106, Revised)

The address of an object with automatic storage **shall not** be assigned to an object which persists after the object has ceased to exist.

Rationale: An object in a function with automatic storage comes into existence when a function is called and disappears when the function is exited. Obviously if the object disappears when the function exits, the address of the object is invalid as well. See Also AV Rule 111 and AV Rule 112.

AV Rule 174 (MISRA Rule 107)

The null pointer **shall not** be de-referenced.

Rationale: De-referencing a NULL pointer constitutes undefined behavior. [10] Note that this often requires that a pointer be checked for non-NULL status before de-referencing occurs.

AV Rule 175

A pointer **shall not** be compared to NULL or be assigned NULL; use plain 0 instead.

Rationale: The NULL macro is an implementation-defined C++ null pointer constant that has been defined in multiple ways including 0, 0L, and (void*)0. Due to C++'s stronger type-checking, Stroustrup[2] advises the use plain 0 rather than any suggested NULL macro.

AV Rule 176

A typedef **will** be used to simplify program syntax when declaring function pointers.

Rationale: Improved readability. Pointers to functions can significantly degrade program readability.

4.23 Type Conversions

AV Rule 177

User-defined conversion functions **should** be avoided. See Meyers [7], item 5.

Rationale: User-defined conversion functions may be called implicitly in cases where the programmer may not expect them to be called. See AV Rule 177 in Appendix A for additional details.

AV Rule 178

Down casting (casting from base to derived class) **shall** only be allowed through one of the following mechanism:

- Virtual functions that act like dynamic casts (most likely useful in relatively simple cases)
- Use of the visitor (or similar) pattern (most likely useful in complicated cases)

Rationale: Casting from a base class to a derived class is unsafe unless some mechanism is provided to ensure that the cast is legitimate.

Note: *Type fields* **shall not** be used as they are too error prone.

Note: Dynamic casts are not allowed at this point due to lack of tool support, but could be considered at some point in the future after appropriate investigation has been performed for SEAL1/2 software. Dynamic casts are fine for general purpose software.

AV Rule 179

A pointer to a virtual base class **shall not** be converted to a pointer to a derived class.

Rationale: Since the *virtualness* of inheritance is not a property of a base class, the layout of a derived class object, referenced through a virtual base pointer, is unknown at compile time. In essence, this type of downcast cannot be performed safely without the use of a *dynamic_cast* or through virtual functions emulating a *dynamic_cast*.

AV Rule 180 (MISRA Rule 43)

Implicit conversions that may result in a loss of information **shall not** be used.

Rationale: The programmer may be unaware of the information loss. See AV Rule 180 in Appendix A for examples.

Note: Templates can be used to resolve many type conversion issues. Also, any compiler flags that result in warnings for value-destroying conversions should be activated.

AV Rule 181 (MISRA Rule 44)

Redundant explicit casts **will not** be used.

Rationale: Unnecessary casting clutters the code and could mask later problems if variable types change over time.

AV Rule 182 (MISRA Rule 45)

Type casting from any type to or from pointers **shall not** be used.

Rationale: This type of casting can lead to undefined or implementation-defined behavior (e.g. certain aspects of memory alignments are implementation-defined). Furthermore, converting a pointer to an integral type can result in the loss of information if the pointer can represent values larger than the integral type to which it is converted.

Exception 1: Casting from *void** to *T** is permissible. In this case, *static_cast* should be used, but only if it is known that the object really is a *T*. Furthermore, such code should only occur in low level memory management routines.

Exception 2: Conversion of literals (i.e. hardware addresses) to pointers.

Device_register input = reinterpret_cast<Device_register>(0XFFA);

AV Rule 183

Every possible measure **should** be taken to avoid type casting.

Rationale: Errors caused by casts are among the most pernicious, particularly because they are so hard to recognize. Strict type checking is your friend – take full advantage of it.

AV Rule 184

Floating point numbers **shall not** be converted to integers unless such a conversion is a specified algorithmic requirement or is necessary for a hardware interface.

Rationale: Converting a floating-point number to an integer may result in an overflow or loss of precision. It is acceptable to explicitly cast integers to floating point numbers to perform mathematical operations (with awareness of the possible real-time impacts as well as overflow). If this is necessary, the deviation must clearly state how an overflow condition cannot occur.

AV Rule 185

C++ style casts (const_cast, reinterpret_cast, and static_cast) **shall** be used instead of the traditional C-style casts. See Stroustrup [2], 15.4 and Meyers [7], item 2.

Rationale: C-style casts are more dangerous than the C++ named conversion operators since the C-style casts are difficult to locate in large programs and the intent of the conversion is not explicit (i.e. *(T) e* could be a portable conversion between related types, a non-portable conversion between unrelated types, or a combination of conversions).[0] See AV Rule 185 in Appendix A for additional details.

4.24 Flow Control Structures

AV Rule 186 (MISRA Rule 52)

There **shall** be no unreachable code.

Note: For *reusable template components*, unused members will not be included in the object code.

AV Rule 187 (MISRA Rule 53, Revised)

All non-null statements **shall** potentially have a side-effect.

Rationale: A non-null statement with no potential side-effect typically indicates a programming error. See AV Rule 187 in Appendix A for additional information.

AV Rule 188 (MISRA Rule 55, Revised)

Labels **will not** be used, except in *switch* statements.

Rationale: Labels are typically either used in switch statements or are as the targets for *goto* statements. See exception given in AV Rule 189.

AV Rule 189 (MISRA Rule 56)

The *goto* statement **shall not** be used.

Rationale: Frequent use of the *goto* statement tends to lead to code that is both difficult to read and maintain.

Exception: A *goto* may be used to break out of multiple nested loops provided the alternative would obscure or otherwise significantly complicate the control logic.

AV Rule 190 (MISRA Rule 57)

The *continue* statement **shall not** be used.

AV Rule 191 (MISRA Rule 58)

The *break* statement **shall not** be used (except to terminate the cases of a *switch* statement).

Exception: The break statement may be used to “break” out of a single loop provided the alternative would obscure or otherwise significantly complicate the control logic.

AV Rule 192 (MISRA Rule 60, Revised)

All *if*, *else if* constructs **will** contain either a final *else* clause or a comment indicating why a final *else* clause is not necessary.

Rationale: Provide a defensive strategy to ensure that all cases are handled by an *else if* series. See AV Rule 192 in Appendix A for examples.

Note: This rule only applies when an *if* statement is followed by one or more *else if*'s.

AV Rule 193 (MISRA Rule 61)

Every non-empty *case* clause in a *switch* statement **shall** be terminated with a *break* statement.

Rationale: Eliminates potentially confusing behavior since execution will *fall through* to the code of the next case clause if a *break* statement does not terminate the previous case clause. See AV Rule 193 in Appendix A for an example.

AV Rule 194 (MISRA Rule 62, Revised)

All *switch* statements that do not intend to test for every enumeration value **shall** contain a final *default* clause.

Rationale: Omitting the final default clause allows the compiler to provide a warning if all enumeration values are not tested in a switch statement. Moreover, the lack of a *default* clause indicates that a test for every case should be conducted. On the other hand, if all cases are not tested for, then a final *default* clause must be included to handle those *untested* cases.

MISRA revised with shall replacing should.

AV Rule 195 (MISRA Rule 63)

A *switch* expression **will not** represent a Boolean value.

Rationale: An *if* statement provides a more natural representation.

AV Rule 196 (MISRA Rule 64, Revised)

Every *switch* statement **will** have at least two *cases* and a potential *default.*

Rationale: An *if* statement provides a more natural representation.

AV Rule 197 (MISRA Rule 65)

Floating point variables **shall not** be used as loop counters.

Rationale: Subjects the loop counter to rounding and truncation errors.

AV Rule 198

The initialization expression in a *for* loop **will** perform no actions other than to initialize the value of a single *for* loop parameter. Note that the initialization expression may invoke an accessor that returns an initial element in a sequence:

```
for (Iter_type p = c.begin() ; p != c.end() ; ++p)          // Good
{
…
}
```

Rationale: Readability.

AV Rule 199

The increment expression in a *for* loop **will** perform no action other than to change a single loop parameter to the next value for the loop.

Rationale: Readability.

AV Rule 200

Null initialize or increment expressions in *for* loops **will not** be used; a *while* loop **will** be used instead.

Rationale: A *while* loop provides a more natural representation.

AV Rule 201 (MISRA Rule 67, Revised)

Numeric variables being used within a *for* loop for iteration counting **shall not** be modified in the body of the loop.

Rationale: Readability and maintainability.

MISRA Rule 67 was revised by changing should to shall.

4.25 Expressions

AV Rule 202 (MISRA Rule 50)

Floating point variables **shall not** be tested for exact equality or inequality.

Rationale: Since floating point numbers are subject to rounding and truncation errors, exact equality may not be achieved, even when expected.

AV Rule 203 (MISRA Rule 51, Revised)

Evaluation of expressions **shall not** lead to overflow/underflow (unless required algorithmically and then should be heavily documented).

Rationale: Expressions leading to overflow/underflow typically indicate overflow error conditions. See also AV Rule 212.

AV Rule 204

A single operation with side-effects **shall** only be used in the following contexts:

1. by itself
2. the right-hand side of an assignment
3. a condition
4. the only argument expression with a side-effect in a function call
5. condition of a loop
6. switch condition
7. single part of a chained operation.

Rationale: Readability. See AV Rule 204 in Appendix A for examples.

AV Rule 204.1 (MISRA Rule 46)

The value of an expression **shall** be the same under any order of evaluation that the standard permits.

Rationale: Except where noted, the order in which operators and subexpression are evaluated, as well as the order in which side effects take place, is unspecified [10], 5(4). See AV Rule 204.1 in Appendix_A for examples.

AV Rule 205

The *volatile* keyword **shall not** be used unless directly interfacing with hardware.

Rationale: The volatile keyword is a hint to the compiler that an object's value may change in ways not specified by the language (e.g. object representing a hardware register). Hence, aggressive optimizations should be avoided. [2]

4.26 Memory Allocation

AV Rule 206 (MISRA Rule 118, Revised)

Allocation/deallocation from/to the free store (heap) **shall not** occur after initialization.

Note that the "placement" *operator new()*, although not technically dynamic memory, may only be used in low-level memory management routines. See AV Rule 70.1 for object lifetime issues associated with placement *operator new()*.

Rationale: repeated allocation (new/malloc) and deallocation (delete/free) from the free store/heap can result in free store/heap fragmentation and hence non-deterministic delays in free store/heap access. See Alloc.doc for alternatives.

AV Rule 207

Unencapsulated global data **will** be avoided.

Rationale: Global data is dangerous since no access protection is provided with respect to the data.

Note: If multiple clients require access to a single resource, that resource should be wrapped in a class that manages access to that resource. For example, semantic controls that prohibit unrestricted access may be provided (e.g. singletons and input streams). See AV_Rule_207_Appendix_A for examples.

4.27 Fault Handling

AV Rule 208

C++ exceptions **shall not** be used (i.e. *throw*, *catch* and *try* shall not be used.)

Rationale: Tool support is not adequate at this time.

4.28 Portable Code

4.28.1 Data Abstraction

AV Rule 209 (MISRA Rule 13, Revised)

The basic types of *int*, *short*, *long*, *float* and *double* **shall not** be used, but specific-length equivalents should be *typedef*'d accordingly for each compiler, and these type names used in the code.

Rationale: Since the storage length of types can vary from compiler to compiler and platform-to-platform, this rule ensures that code can be easily reconfigured for storage size differences by simply changing definitions in one file. See AV Rule 209 in Appendix A for additional details.

Exception: Basic types are permitted in low-level routines to assist in the management of word alignment issues (e.g. memory allocators).

MISRA rule was changed from should to shall.

4.28.2 Data Representation

AV Rule 210

Algorithms **shall not** make assumptions concerning how data is represented in memory (e.g. big endian vs. little endian, base class subobject ordering in derived classes, nonstatic data member ordering across access specifiers, etc.)

Rationale: Assumptions concerning architecture-specific aspects are non-portable.

Exception: Low level routines that are expressly written for the purpose of data formatting (e.g. marshalling data, endian conversions, etc.) are permitted.

AV Rule 210.1

Algorithms **shall not** make assumptions concerning the order of allocation of nonstatic data members separated by an access specifier. See also AV Rule 210 on data representation.

Rationale: The order of allocation of nonstatic data members, separated by an access-specifier, is unspecified [10], 9.2(12). See AV Rule 210.1 in Appendix_A for additional details.

AV Rule 211

Algorithms **shall not** assume that *shorts*, *ints*, *longs*, *floats*, *doubles* or *long doubles* begin at particular addresses.

Rationale: The representation of data types in memory is highly machine-dependent. By allocating data members to certain addresses, a processor may execute code more efficiently. Because of this, the data structure that represents a structure or class will sometimes include holes and be stored differently in different process architectures. Code which depends on a specific representation is, of course, not portable.

Exception: Low level routines that are expressly written for the purpose of data formatting (e.g. marshalling data, endian conversions, etc.) are permitted.

4.28.3 Underflow/Overflow

AV Rule 212

Underflow or overflow functioning **shall not** be depended on in any special way.

Rationale: Dependence on undefined language aspects leads to non-portable implementations. See also AV Rule 203.

4.28.4 Order of Execution

AV Rule 213 (MISRA Rule 47, Revised)

No dependence **shall** be placed on C++'s operator precedence rules, below arithmetic operators, in expressions.

Rationale: Readability. See AV Rule 213 in Appendix A for additional details.

MISRA Rule 47 changed by replacing should with shall.

AV Rule 214

Assuming that non-local static objects, in separate translation units, are initialized in a special order **shall not** be done.

Rationale: Order dependencies lead to hard to find bugs. See AV Rule 214 in Appendix A for additional details.

4.28.5 Pointer Arithmetic

AV Rule 215 (MISRA Rule 101)

Pointer arithmetic **will not** be used.

Rationale: The runtime computation of pointer values is error-prone (i.e. the computed value may reference unintended or invalid memory locations). See AV Rule 97 and AV Rule 215 in Appendix A for additional information.

Exceptions: Objects such as containers, iterators, and allocators that manage pointer arithmetic through well-defined interfaces are acceptable.

4.29 Efficiency Considerations

AV Rule 216

Programmers **should not** attempt to prematurely optimize code. See Meyers [7], item 16.

Rationale: Early focus on optimization can result in sacrificing the clarity and generality of modules that will not be the true bottlenecks in the final system. See AV Rule 216 in Appendix A for additional details.

Premature optimization is the root of all evil – Donald Knuth

Note: This rule does not preclude early consideration of fundamental algorithmic and data structure efficiencies.

See also AV Rule 125 and AV Rule 177 for performance recommendations.

4.30 Miscellaneous

AV Rule 217

Compile-time and link-time errors **should** be preferred over run-time errors. See Meyers [6], item 46.

Rationale: Errors detected at compile/link time will not occur at run time.

Whenever possible, push the detection of an error back from run-time to link-time, and preferably compile-time. See also AV Rule 103 and AV Rule 194.

AV Rule 218

Compiler warning levels **will** be set in compliance with project policies.

Rationale: Compilers can typically be configured to generate a useful set of warning messages that point out potential problems. Information gleaned from these messages could be used to resolve certain errors before they occur at runtime.

5 TESTING

This section provides guidance when testing inheritance hierarchies that employ virtual functions.

5.1.1 Subtypes

If *D* is a subtype of *B*, then instances of type *D* will function transparently in any context in which instances of type *B* can exist. Thus it follows that all base class unit-level test cases must be inherited by the test plan for derived classes. That is, derived classes must at least successfully pass the test cases applicable to their base classes.[3]

AV Rule 219

All tests applied to a base class interface **shall be** applied to all derived class interfaces as well. If the derived class poses stronger postconditions/invariants, then the new postconditions /invariants shall be substituted in the derived class tests.

Rationale: A publicly-derived class must function transparently in the context of its base classes.

Note: This rule will often imply that every test case appearing in the set of test cases associated with a class will also appear in the set of test cases associated with each of its derived classes.

5.1.2 Structure

AV Rule 220

Structural coverage algorithms **shall** be applied against *flattened* classes.

Rationale: Structural coverage reporting should be with respect to each class context—not a summed across multiple class contexts. See AV Rule 220 in Appendix A for additional details.

Note: When a class is viewed with respect to all of its components (both defined at the derived level as well as inherited from all of its base levels) it is said to be *flattened*.

AV Rule 221

Structural coverage of a class within an inheritance hierarchy containing virtual functions **shall** include testing every possible resolution for each set of identical polymorphic references.

Rationale: Provide decision coverage for dispatch tables.

[3] Note that subclass tests will often be extensions of the superclass tests.

APPENDIX A

AV Rule 3

Cyclomatic complexity measures the amount of decision logic in a single software module. It may be used for two related purposes: a measure of design complexity and an aid in testing. First, cyclomatic complexity may be utilized throughout all phases of the software lifecycle, beginning with design, to enhance software reliability, testability, and maintainability. Second, cyclomatic complexity aids in the test planning process by approximating the number of tests required for a given module. Cyclomatic complexity is a structural metric based entirely on control flow through a piece of code; it is the number of non-repeating paths through the code.

Cyclomatic complexity (*v(G)*) is defined for each module to be:

v(G) = e - n + 2

where *n* represents 'nodes', or computational statements, and *e* represents 'edges', or the transfer of control between nodes.

Below is an example of source code followed by a corresponding node diagram. In the node diagram, statements are illustrated as rectangles, decisions as triangles and transitions between statements as lines. The number of nodes is fourteen while the number of lines connecting the nodes is seventeen for a complexity of five.

Another means of estimating complexity is also illustrated. The number of regions bounded by the lines, including the "infinite" region outside of the function, is generally equivalent to the computed complexity. The illustration has 5 disjoint regions; note that it is equal to the computed complexity.

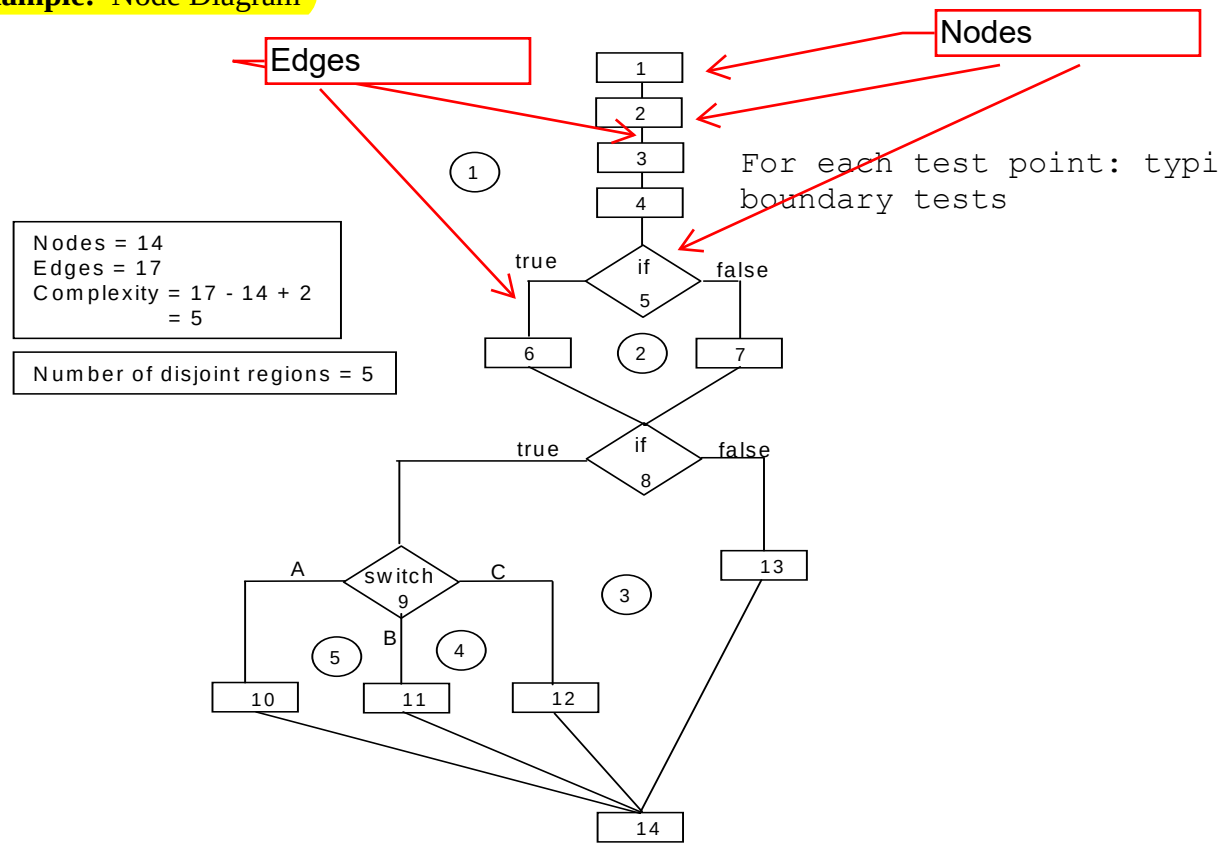
The illustration uses a multi-way decision or switch statement. Often, a switch statement may have many cases causing the complexity to be high, yet the code is still easy to comprehend. Therefore, complexity limits should be set keeping in mind the ultimate goals: sensible and maintainable code.

Example: Source Code

```
void compute_pay_check ( employee_ptr_type      employee_ptr_IP,
                         check_ptr_type         chk_ptr_OP )
{
        //Calculate the employee's federal, fica and state tax withholdings
1.      chk_ptr_OP->gross_pay   = employee_ptr_IP->base_pay;
2.      chk_ptr_OP->ged_tax     = federal_tax ( employee_ptr_IP->base_pay );
3.      chk_ptr_OP->fica        = fica        ( employee_ptr_IP->base_pay );
4.      chk_ptr_OP->state_tax   = state_tax   ( employee_ptr_IP->base_pay );

        //Determine medical expense based on the employee's HMO selection
5.      if ( employee_ptr_IP->participate_HMO == true )
```

```
        {
6.        chk_ptr_OP->medical = med_expense_HMO;
         }
         else
         {
7.        chk_ptr_OP->medical = med_expense_non_HMO;
         }

         // Calc a profit share deduction based on % of employee's gross pay
8.       if (employee_ptr_IP->participate_profit_share == true )
        {
9.        switch( employee_ptr_IP->profit_share_plan )
          {
            case plan_a:
10.           chk_ptr_OP->profit_share = two_percent * chk_ptr_OP->gross_pay;
              break;
            case plan_b:
11.           chk_ptr_OP->profit_share = four_percent * chk_ptr_OP->gross_pay;
              break;
            case plan_c:
12.           chk_ptr_OP->profit_share = six_percent * chk_ptr_OP->gross_pay;
              break;
            default:
              break;
          }
        }
        else
        {
13.       chk_ptr_OP->profit_share = zero;
         }

        chk_ptr_OP->net_pay = ( chk_ptr_OP->gross_pay –
                                chk_ptr_OP->fed_tax –
                                chk_ptr_OP->fica –
                                chk_ptr_OP->state_tax –
                                chk_ptr_OP->medical –
                                chk_ptr_OP->profit_share );
}
```

Example: Node Diagram



AV Rule 11

Trigraphs can lead to confusion when question marks are used. For example, the string:

"Enter the date in the following form (??-??-????)"

would be interpreted as

"Enter the date in the following form (~~??]"

AV Rule 12

The use of digraphs listed in this rule can obscure the meaning of otherwise simple constructs. For example,

```
int16    a <: 2 :> <: 2 :> = <%<%0,1%>,<%2,3%>%>;
```

is more simply written as

```
int16    a[2][2] = { {0,1}, {2,3} };
```

AV Rule 15

For SEAL 1/2 applications, defensive programming checks are required. Defensive programming is the practice of evaluating potential failure modes (due to hardware failures and/or software errors) and providing safeguards against those failure modes. For SEAL 1/2 software, System Safety is required to define all the possible software hazards (conditions in which software could contribute to the loss of system function). If the determination is made from the system level that hazard mitigation will be in software, then software requirements must be derived (from the identified software hazards) to define appropriate hazard mitigations. During coding and subsequent code inspection, the code must be evaluated to ensure that the defensive programming techniques implied by the hazard mitigation requirements have been implemented and comply with the requirements. Examples where defensive programming techniques are used include (but are not limited to) management of:

- arithmetic errors—Overflow, underflow, divide-by-zero, etc. (See also AV Rule 203)
- pointer arithmetic errors—A dynamically calculated pointer references an unreasonable memory location. (See also AV Rule 215)
- array bounds errors—An array index does not lie within the bounds of the array. (See also AV Rule 97)
- range errors—Invalid arguments passed to functions (e.g. passing a negative value to the *sqrt()* function).

Note that explicit checks may not be required in all cases, but rather some other form of analysis may be used that achieves the same end. Consider, for example, the following use of container *a*. Notice that bounds errors are not possible by construction. Hence, array-access bounds errors are managed without explicit checks.

```
const uint32 n = a.size();
for (uint32 i=0 ; i<n ; ++i)
{
   a[i] = i;

}
```

AV Rule 29

Inline functions do not require text substitutions and are well-behaved when called with arguments (e.g. type-checking is performed).

Example: Compute the maximum of two integers.

```
#define max (a,b) ((a > b) ? a : b)        // Wrong: macro

inline int32   maxf (int32 a, int32 b)     // Correct: inline function
{
   return (a > b) ? a : b;
}

y = max (++p,q);                           // Wrong: ++p evaluated twice

y=maxf (++p,q)                             // Correct: ++p evaluated once and type
                                           // checking performed. (q is const)
```

AV Rule 30

Since *const* variables follow scope rules, are subject to type checking, and do not require text substitutions (which can be confusing or misleading), they are preferable to macros as illustrated in the following example.

Example:

```
#define max_count          100          // Wrong: no type checking
const int16 max_count   = 100;          // Correct: type checking may be performed
```

Note: Integral constants can be eliminated by optimizers, but non-integral constants will not. Thus, in the example above, *max_count* will not be laid down in the resulting image.

AV Rule 32

The exception to the rule involves template class and function definitions which may be partitioned into separate header and implementation files. In this case, the implementation file may be included as a part of the header file. Note that the implementation file is logically part of the header and is not separately compilable as illustrated below.

Example:

File A.h:

```
--------------------------------
#ifndef A_H
#define A_H

template< class T >
class A
{
public:
   void do_something();
};

#include <A.cpp>
#endif
--------------------------------
```

File A.cpp:

```
--------------------------------
template< class T >
A<T>::do_something()
{
   // do_something impelemtation
}
--------------------------------
```

AV Rule 36

Unnecessary recompilation of source files should be eliminated when possible. In the following example, each source file includes all header files without a determination of which ones are actually required.

Example: All header files are included in the three source files regardless of which files are actually required. This creates several problems:

1. Inability to limit compilation scope. That is, any change to one header file means recompiling (and consequently retesting) each source file.
2. Unnecessarily long compilation times. The repeated compilation of unnecessary header files will significantly increase the overall compilation time.

```
// File 1
#include <header1.h>
#include <header2.h>   // Incorrect: unneeded
#include <header3.h>   // Incorrect: unneeded
…                      // Source for file 1

// File 2
#include <header1.h>   // Incorrect: unneeded
#include <header2.h>
#include <header3.h>   // Incorrect: unneeded
…                      // Source for file 2

// File 3
#include <header1.h>   // Incorrect: unneeded
#include <header2.h>   // Incorrect: unneeded
#include <header3.h>
…                      // Source for file 3
```

AV Rule 38

The header files of classes that are only referenced via pointers or references need not be included. Doing so often increases the coupling between classes, leading to increased compilation dependencies as well as greater maintenance efforts. *Forward declarations* of the classes in question (supplied by *forward headers*) can be used to limit implementation dependencies, maintenance efforts and compile times.

Example A: This example unnecessarily includes header files creating additional dependencies in the Operator interface.

```
// Operator.h
#include <LM_string.h>          // Incorrect: creates unnecessary dependency
#include <Date.h>               // Incorrect: creates unnecessary dependency
#include <Record.h>             // Incorrect: creates unnecessary dependency

class Operator
{
 public:
   Operator (const LM_string &name,
             const Date      &birthday,
             const Record    &flying_record);

   LM_string    get_name () const;
   int32        get_age () const;
   Record       get_record () const;
   …
 private:
   Operator_impl *impl;
};
```

Example B: In contrast to Example A, Example B uses *forward headers* to forward declare implementation classes used by Operator. Hence the Operator interface is not dependent on any of the implementation classes.

```
// Operator.h                          The forward headers only contain declarations.
#include        <LM_string_fwd.h>
#include        <Date_fwd.h>
#include        <Record_fwd.h>
#include        <OperatorImpl.h>

class Operator
{
  public:
    Operator (const LM_string    &name,
              const Date         & birthday,
              const Record       &flying_record);

    LM_string   get_name() const;
    int32       get_age   () const;
    record      get_record () const;
```

```
    …
 private:
   Operator_impl *impl;
};

// Operator.cc
#include   <Operator.h>
#include   <Operator_impl.h>   // Contains implementation details of the Operator object.
…
int32 Operator::get_age()
{
   impl->get_age();
}
```

AV Rule 39

Although header files should not contain non-const variable or function definitions in general, inline functions and template definitions will often be included.

Example: Although definitions should, in general, be placed in .cpp files, a member function defined inside a class declaration represents a suggestion to the compiler that the member function should be inlined (if possible).

```
class Square
{
 public:
    float32   area()              // The member function definition in the class declaration
   {                              // suggests to the compiler that the member function should be
       return length  *width;     // inlined.
   }

 private:
   float32    length;
   float32   width;
};
```

AV Rule 40

AV Rule 40 is intended to support the one definition rule (ODR). That is, only a single definition for each entity in a program may exist. Hence, placing the declaration of a type (included with its definition) in a single header file ensures that duplicate definitions are identical.

Example A: Scattering the definition of a type throughout an application (i.e. in .cpp files) increases the likelihood of non-unique definitions (i.e. violations of the ODR).

```
//s.cpp
class S                 // Bad:  S declared in .cpp file.
{                       //       S could be declared differently in a
        int32   x;      //       separate .cpp file
        char    y;
};
```

Example B: Placing the definition of S in two different header files provides an opportunity for non-unique definitions (i.e. violation of the ODR).

```
//s.h
class S
{
        int32   x;
        char    y;
};

// y.h
class S         // Bad: S multiply defined in two different header files.
{
        int32   x;
        int32   y;

};
```

AV Rule 42

AV Rule 42 indicates that expression-statements must be on separate lines. An expression statement has the following form:

expression-statement:
expression$_{opt}$;

All expressions in an expression-statement are evaluated and all side effects are completed before the next statement is executed. The most common expression-statements are assignments and function calls. [10]

Examples:

```
x = 7; y=3;              // Incorrect: multiple expression statements on the same line.
a[i] = j[k]; i++; j++;   // Incorrect: multiple expression statements on the same line.
a[i] = k[j];             // Correct.
i++;
j++;
```

Note that a *for* statement is a special case where *condition*$_{opt}$ and *expression*$_{opt}$ may appear on the same line as *expression-statement*[10].

iteration-statement:
`while` (*condition*) *statement*
`do` *statement* `while` (*expression*) ;
`for` (*for-init-statement condition-opt* ; *expression-opt*) *statement*

for-init-statement:
expression-statement
simple-declaration

Examples:

```
for( i = 0 ; i < max ; ++i) fun(); // Incorrect: multiple expression statements on the same line.
for(i = 0 ; i < max ; ++i)         // Correct
{
   foo();
}
```

AV Rule 58

Examples: The following examples illustrate the proper way to declare functions with multiple arguments.

```
int32 max (int32 a, int32 b)              // Correct: two parameters may appear on the
{                                         // same line. Order is easily understood.
…
}
                                          // Incorrect: too many parameters on the same line.
                                          // Difficult to document parameters in this form
msg1_in (uint16  msg_ID, float32  rate_IO, uint32  msg_size, uint16  rcv_max_instances)
{
…
}

                                          // Correct form.
msg1_in ( uint16   msg_ID,                // Unique identifier that is the label for the message
          float32  rate_IO,               // The desired rate for the message distributed
          uint32   msg_size,              // Size in bytes of the message
          uint16   rcv_max_instances)     // The maximum number of instances of this
                                          // message expected in a processing frame
{
…
}
```

AV Rule 59

As the following examples illustrate, the bodies of *if, else if, else, while, do..while* and *for* statements should always be enclosed within braces. As illustrated in Example A, code added at a later time will not be part of a block unless it is enclosed by braces. Furthermore, as illustrated by Example B, ";" can be difficult to see by itself. Hence a block (even if empty) is required after control flow primitives.

Example A:

```
if (flag == 1)
{
   success ();
}
else                              // Incorrect: log_error() was added at a later time
   clean_up_resources();          //            but is not part of the block (even though
   log_error();                   //            it is at the proper indentation level).
```

Example B: A block, even if empty, is required after control flow primitives.

```
while (f(x));          // Incorrect: ";" is difficult to see.
while (f(x))           // Incorrect: ";" is difficult to see.
   ;
while (f(x))           // Correct
{
}
```

AV Rule 70

AV Rule 70 indicates that friends may be used only when a function or object requires access to the private elements of a class, but is unable to be a member of the class for logical or efficiency reasons. The following three examples illustrate acceptable uses of friends.

Example A: *operator<<()*

Consider *operator<<()* and *operator>>()* where an implicit type conversion on the left-most argument is often required. Since an implicit type conversion on the left-most argument of a function can only be provided through non-member functions, *operator<<()* and *operator>>()* must be implemented as friend functions.

The preferred C++ solution is to declare such functions (that are conceptually part of the public interface) as non-member friends of the class. This solution provides both private element access as well as implicit type conversions.

Example B: Binary operator overloads (+, -, *, /, etc.)

Consider the example provided by Stroustrup [2]. How can a matrix-vector multiplication operation be provided without exposing the internal representation of the matrix or the vector? Clearly, the function requires access to the internal representation of both the matrix and the vector. Thus, the function cannot be a member of either one. However, if the function is not a friend, then accessors and mutators must be supplied which expose the internal representation (i.e. violate encapsulation). Hence, adding the friend function *operator*()* to the public interface of both *Matrix* and *Vector* provides a clean, encapsulated approach.

```
class Matrix;

class Vector
{
   float32  v[4];
   // …
   friend Vector  operator* (const matrix& m,  const vector& v);
};

class Matrix {
   Vector v[4];
   // …
   friend Vector operator*(const Matrix& m, const Vector& v);
};

Vector operator*(const Matrix& m, const Vector& v)
{
```

```
    Vector r;

    for (int32   i=0 ; i<4 ; i++)
    {
       r.v[i] = 0;
       for(int32   j=0; j<4 ; j++)
      {
          r.v[i] += m.v[i].v[j] * v.v[j];
       }
    }

    return r;
}
```

Example C: External iterators.

Since an iterator may be required to modify the contents of an object within a container (*iterator = value), it must be able to access the private portions of that object. Thus, if an iterator is external to a class, it must be a friend.

AV Rule 70.1

Conceptually, developers understand that objects should not be used before they have been created or after they have been destroyed. However, a number of scenarios may arise where this distinction may not be obvious. Consequently, a series of examples is provided to highlight possible areas of confusion. In many cases, the C++ standard [10] is quoted and an explanatory code segment is provided.

Example A: Exiting *main()*.

main() should never exit independent of the application of which it is a part. Consider the code sample below. When *main()* exits, the static object destructors are invoked. Hence, the tasks created by *main()* cannot depend the existence those static objects.

```
int32 main()
{
   _main();                  // Call static constructors (inserted by compiler)

// Application code begins
   initialize_task_1();      // Initialize tasks
   initialize_task_2();
   …
   initialize_task_n();
// Application code ends

   __call_dtors();           // Call static destructors (inserted by compiler)
}

// Tasks begin to run. However, static objects have been destroyed.
```

Example B: Accessing a const Object During Construction.

Note that this scenario cannot occur without the use of global variables which are prohibited by AV Rule 207.

> During the construction of a const object, if the value of the object or any of its subobjects is accessed through an lvalue that is not obtained, directly or indirectly, from the constructor's this pointer, the value of the object or subobject thus obtained is unspecified. [10] 12.1(15)

```
struct C;
void no_opt(C*);
struct C
{
   int c;
   C() : c(0)
   {
      no_opt(this);
   }
};

const C cobj;
void no_opt(C* cptr)
{
   int i = cobj.c * 100          // value of cobj.c is unspecified
   cptr->c = 1;
   cout << cobj.c * 100          // value of cobj.c is unspecified
   << '\n';
}
```

Example C: Local Static Object with Non-Trivial Destructors.

If a function contains a local object of static storage duration that has been destroyed and the function is called during the destruction of an object with static storage duration, the program has undefined behavior if the flow of control passes through the definition of the previously destroyed local object. [10] 3.6.3(2)

```
class A
{
   public:
      ~A() { … }
};

void foo()
{
   static A a;              // Destructor of local static will be invoked on exit
}

class B
{
   public:
      ~B()
       {
          foo();            // Destructor of static calls function with local static which may
        }                   // already be destroyed.
};
static B B_var;             // Destructor of static will be invoked on exit.
```

Example D: Invocation of Member Function after Lifetime of Object has Ended.

Before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that refers to the storage location where the object will be or was located may be used but only in limited ways. …if the object will be or was of a non-POD class type, the program has undefined behavior if:

- the pointer is used to access a non-static data member or call a non-static member function of the object, ... [10] 3.8(5)

```
struct B
{
   virtual void f();
   void mutate();
   virtual ~B();
 };

struct D1 : B
{
   void f()
};

struct D2 : B
{
    void f()
};

void B::mutate()
{
   new (this) D2;     // reuses storage – ends the lifetime of *this
   f();               // undefined behavior
   ... = this;        // OK, this points to valid memory
}
                      // Note: placement new is only allowed in low-level memory
                      //       management routines (see AV Rule 206).
```

Example E: Storage Reuse does not Require Implicit Destructor Invocation.

For an object of a class type with a non-trivial destructor, the program is not required to call the destructor explicitly before the storage which the object occupies is reused or released; however, if there is no explicit call to the destructor or if a delete-expression (5.3.5) is not used to release the storage, the destructor shall not be implicitly called and any program that depends on the side effects produced by the destructor has undefined behavior. [8] 3.8(4)

```
struct A
{
   ~A()
   {
      …non-trivial destructor
   }
};

struct B { … };

void c_03_06_driver()
{
   A a_obj;
   new (&a_obj) B();     // a_obj's lifetime ended without calling
   …                     // nontrivial destructor.
}
                         // Note: placement new is only allowed in low-level memory
                         //       management routines (see AV Rule 206).
```

Example F: Object of Original Type Must Occupy Storage for Implicit Destructor Call.

If a program ends the lifetime of an object of type T with static (3.7.1) or automatic (3.7.2) storage duration and if T has a non-trivial destructor, the program must ensure that an object of the original type occupies that same storage location when the implicit destructor call takes place; otherwise the behavior of the program is undefined. This is true even if the block is exited with an exception. [10] 3.8(8)

```
class T { };

struct B {
   ~B() { … };
};

void c_03_11_driver()
{
  B b;
  new (&b) T;          // B's nontrivial dtor implicitly called on memory occupied by an
                       // object of different type.
}                       //undefined behavior at block exit

                       // Note: placement new is only allowed in low-level memory
                       //       management routines (see AV Rule 206).
```

Example G: Creating a New Object at the Storage Location of a const Object.

Creating a new object at the storage location that a const object with static or automatic storage duration occupies or, at the storage location that such a const object used to occupy before its lifetime ended results in undefined behavior. [10] 3.8(9)

```
struct B
{
   B() { … };
   ~B() { … };
};
const B b;
void c_03_12_driver()
{
   b.~B();              // A new object is created at the storage location that a const
                        // object used to occupy before its lifetime ended. This results
   new (&b) const B;    // in undefined behavior
}
                        // Note: placement new is only allowed in low-level memory
                        //       management routines (see AV Rule 206).
```

Example H: Member Function in ctor-Initializer Invoked Before Bases are Initialized.

If these operations (member function invocation, operand of typeid or dynamic_cast) are performed in a ctor-initializer (or in a function called directly or indirectly from a ctor-initializer) before all the mem-initializers for base classes have completed, the result of the operation is undefined. [10] 12.6.2(8)

```
class A { public: A(int) { … }};

class B : public A
{
    int j;
    public:
       int f() { … };
       B() : A(f()),              // Undefined: calls member function but base A is
                                  // is not yet initialized
             j(f()) { … }         // Well-defined: bases are all initialized
};
```

AV Rule 71

The intent of AV Rule 71 is to prevent an object from being used before it is in a fully initialized state. This may occur in three cases:

1. a class constructor invokes an overridden method before the derived class (supplying the method) has been fully constructed,
2. a class constructor invokes a public or protected method that requires the object to be fully initialized as a pre-condition of method invocation, or
3. the constructor does not fully initialize the object allowing clients access to uninitialized data.

In the first case, C++ will not allow overridden methods to resolve to their corresponding subclass versions since the subclass itself will not have been fully constructed and thus, by definition, will not exist. In other words, while the base class component of a derived class is being constructed, no methods of the derived class can be invoked through the *virtual method* mechanism. Consequently, constructors should make no attempt to employ dynamic binding in any form.

Secondly, public (and in some cases protected) methods assume object initialization and class invariants have been established prior to invocation. Thus, invocation of such methods during object construction risks the use of uninitialized or invalid data since class invariants can not be guaranteed before an object is fully constructed.

Finally, the constructor should fully initialize an object (see Stroustrup [2], Appendix E and AV Rule 72). If for some reason the constructor cannot fully initialize an object, some provision must be made (and documented in the constructor) to ensure that clients cannot access the uninitialized portions of the object.

AV Rule 71.1

The intent of AV Rule 71.1 is to clarify that a class's virtual functions are resolved statically in any of its constructors or its destructor. As a result, the placement of virtual functions in constructors/destructors often leads to unexpected behavior.

Consider the examples below. In Example A, the virtual function does not exhibit polymorphic behavior. In contrast, the same function is called in Example B. This time, however, the scope resolution operator is used to clarify that the virtual function is statically bound.

Example A:

```
class Base
{
   public:
         Base()
         {
            v_fun();              // Bad: virtual function called from constructor. Polymorphic
         }                        //      behavior will not be realized.
         virtual void v_fun()
         {
         }
};
```

Example B:

```
class Base
{
   public:
         Base()
         {
            Base::v_fun();        // Good: scope resolution operator used to specify static
         }                        //       binding
         virtual void v_fun()
         {
         }
};
```

AV Rule 73

A default constructor is a constructor that can be called without any arguments. Calling a constructor without any arguments implies that objects can be created and initialized without supplying external information from the point of call. Although this may be appropriate for some classes of objects, there are others for which there is no reasonable means of initialization without passing in external information. For this class of objects, the presence of default constructors requires that additional logic be added to member functions to ensure complete object initialization before operations are allowed to proceed. Hence, avoiding gratuitous default constructors leads to less complex, more efficient operations on fully initialized objects.

Consider the following examples where a *Part* must always have a *SerialNumber*. *Example A* illustrates the code for a single method, *getPartName()*, that returns the name of the part identified by a particular serial number. Note that additional logic must be added to the member function *getPartName()* to determine if the part has been fully initialized. In contrast, *Example B* does not have the unnecessary default constructor. The corresponding implementation is cleaner, simpler, and more efficient.

Example A: Gratuitous default constructor.

```
class Part
{
  public:
    Part ()
    { serial_number =unknown;
    }                                   // Default constructor:

    Part (int32  n) : serial_number(n) {}
    int32  get_part_name()
    {
      if (serial_number == unknown)             // Logic must be added to check for
      {                                         // uninitialized state
        return “”;
      }
      else
      {
        return lookup_name (serial_number);
       }
  private:
    int32                 serialNumber;
    static const int32    unknown;
};
```

Example B: No gratuitous default constructor.

```
class Part
{
  public:
    Part (int32 n) : serial_number(n) {}
    int32   get_part_name () { return lookup_name (serial_number);}
  private:
```

```
   int32  serial_number;
}
;
```

Note: The absence of a default constructor implies certain restrictions for arrays and template-based containers of such objects. See Meyers [7] for more specific details.

AV Rule 74

Rationale: This rule stems from the following observations:

- Member initialization is the only option for const members.
- Member initialization is the only option for reference members.
- Member initialization is never less efficient and often more efficient than assignment.
- Member initialization tends to simplify maintenance of classes.

Example A: For class *Rectangle* with attributes *length* and *width*, the member initialization list should be used to initialize both attributes.

```
Rectangle (float32 length_, float32 width_) : length(length_), width(width_)
{
}
```

Example B: Suppose that *length* and *width* cannot be represented as simple expressions (e.g. they must be read from an input stream). In this case, the member initialization list cannot be used.

```
Rectangle ()
{
   cin >> length >> width;
}
```

AV Rule 76

If an object contains a pointer to a data element, what should happen when that object is copied? Should the pointer itself be copied and thus two different objects reference the same data item, or should the data pointed to be copied? The default behavior is to copy the pointer. This behavior, however, is often not the desired behavior. The solution is to define both the copy constructor and assignment operator for such cases.

If clients should never be able to make copies of an object, then the copy constructor and the assignment operator should be declared private (with no definition). This will prevent clients from calling these functions as well as compilers from generating them.

Finally, a nontrivial destructor typically implies some form of resource cleanup. Hence, that cleanup will most likely need to be performed during an assignment operation.

Note: There are some cases where the default copy and assignment operators do offer reasonable semantics. For example, a function object holding a pointer to a member function (e.g. std::mem_fun_t) may not require non-default behavior. For these cases, see AV Rule 80.

AV Rule 77

A class may contain many data members as well as exist within an inheritance hierarchy. Hence the copy constructor must copy all members (that affect the class invariant), including those in base classes, as in the following example:

```
class Base
{
  public:
     Base  (int32   x) : base_member (x) { }

     Base  (const Base&  rhs) : base_member (rhs.base_member) {}

  private:
    int32  base_member;
};

class Derived : public Base
{
  public:
    Derived (int32 x, int32 y, int32 z) : Base (x),
                                          derived_member_1 (y),
                                          derived_member_2 (z)  { }

    Derived(const Derived& rhs) : Base(rhs),
                                  derived_member_1 (rhs.derived_member_1),
                                  derived_member_2 (rhs.derived_member_2) { }

  private:
    int32   derived_member_1;
    int32   derived_member_2;
};
```

AV Rule 77.1

A particular ambiguity can arise with respect to compiler-supplied, implicit copy constructors as noted in [10] 12.8(4):

If the class definition does not explicitly declare a copy constructor, one is declared implicitly. Thus, for the class definition

```
struct X {
   X(const X&, int);
};
```

a copy constructor is implicitly-declared. If the user-declared constructor is later defined as

```
X::X(const X& x, int i =0) { /* ... */ }
```

then any use of X's copy constructor is ill-formed because of the ambiguity; no diagnostic is required.

AV Rule 79

Releasing resources in a destructor provides a convenient means of resource management, especially in regards to exceptional cases. Moreover, if it is possible that a resource could be leaked, then that resource should be wrapped in a class whose destructor automatically cleans up the resource.

Example A: Stroustrup [2] provides an example based on a file handle. Note that the constructor opens the file while the destructor closes the file. Any possibility that a client may "forget" to cleanup the resource is eliminated.

```
 class File_ptr                          // Raw file pointer wrapped in class to ensure
{                                        // resources are not leaked.
 public:
     File_ptr   (const char *n, const char * a) { p = fopen(n,a); }
     File_ptr   (FILE* pp)                      { p = pp; }

    ~File_ptr ()
    {
       if (p)
      {
        fclose(p)
      };
    }      // Clean up file handle.
 …
 private:
    FILE *p;
};

use_file (const char  *file_name)
{                                  // Client does not have to remember to clean up file handle
  File_ptr  f(fn,"r");             // (impossible to leak file handles).
                                    // use f
}                                  // f goes out of scope so the destructor is called,
                                   // cleaning up the file handle.
```

AV Rule 81

Self-assignment must be handled appropriately by the assignment operator. Example A illustrates a potential problem, whereas Example B illustrates an acceptable approach.

Example A: Although it is not necessary to check for self-assignment in all cases, the following example illustrates a context where it would be appropriate.

```
Base &operator= (const Base  &rhs)
{
   release_handle (my_handle);          // Error: the resource referenced by myHandle is
   my_handle = rhs.myHandle;            // erroneously released in the self-assignment case.
   return *this;
}
```

Example B: One means of handling self-assignment is to check for self-assignment before further processing continues as illustrated below.

```
Base &operator= (const Base& rhs)
{
  if (this != &rhs)                     // Check for self assignment before continuing.
  {
     release_handle(my_handle);         // Release resource.
     my_handle = rhs.my_handle;                 // Assign members (only one member in class).
  }
  else
  {
  }
  return *this;
}
```

AV Rule 83

A class may contain many data members as well as exist within an inheritance hierarchy. Hence the assignment operator must assign all members, including those in base classes, which affect the class invariant as in the following example:

Note: Definition of *operator=()* is included in the class declaration to simplify the explanation of this rule. It breaks the "no function definition in class declaration" rule.

```
class Base
{
 public:
   Base (int32 x) : base_member (x) {}

   Base &operator=(const Base& rhs)
   {
      if (this != &rhs)                    // Check for self assignment before continuing.
      {
        base_member = rhs.base_member;     // Assign members (only one member in class).
      }
      else
      {
      }
      return *this;
   }

 private:
   int32  base_member;
};

class Derived : public Base
{
 public:
   Derived (int32  x, int32  y, int32  z) : Base (x),
                                            derived_member_1 (y),
                                           derived_member_2 (z) {}

   Derived&  operator=(const Derived&  rhs)
   {
     if (this != &rhs)                                 // Check for self-assignment
     {
       Base::operator=(rhs);                           // Copy base class elements.
       derived_member_1 = rhs.derived_member_1;        // Assign all members of derived class
       derived_member_2 = rhs.derived_member_2;
     }
     else
    {
     }

     return *this;
   }
```

```
 private:
   int32  derived_member_1;
   int32  derived_member_2;
};
```

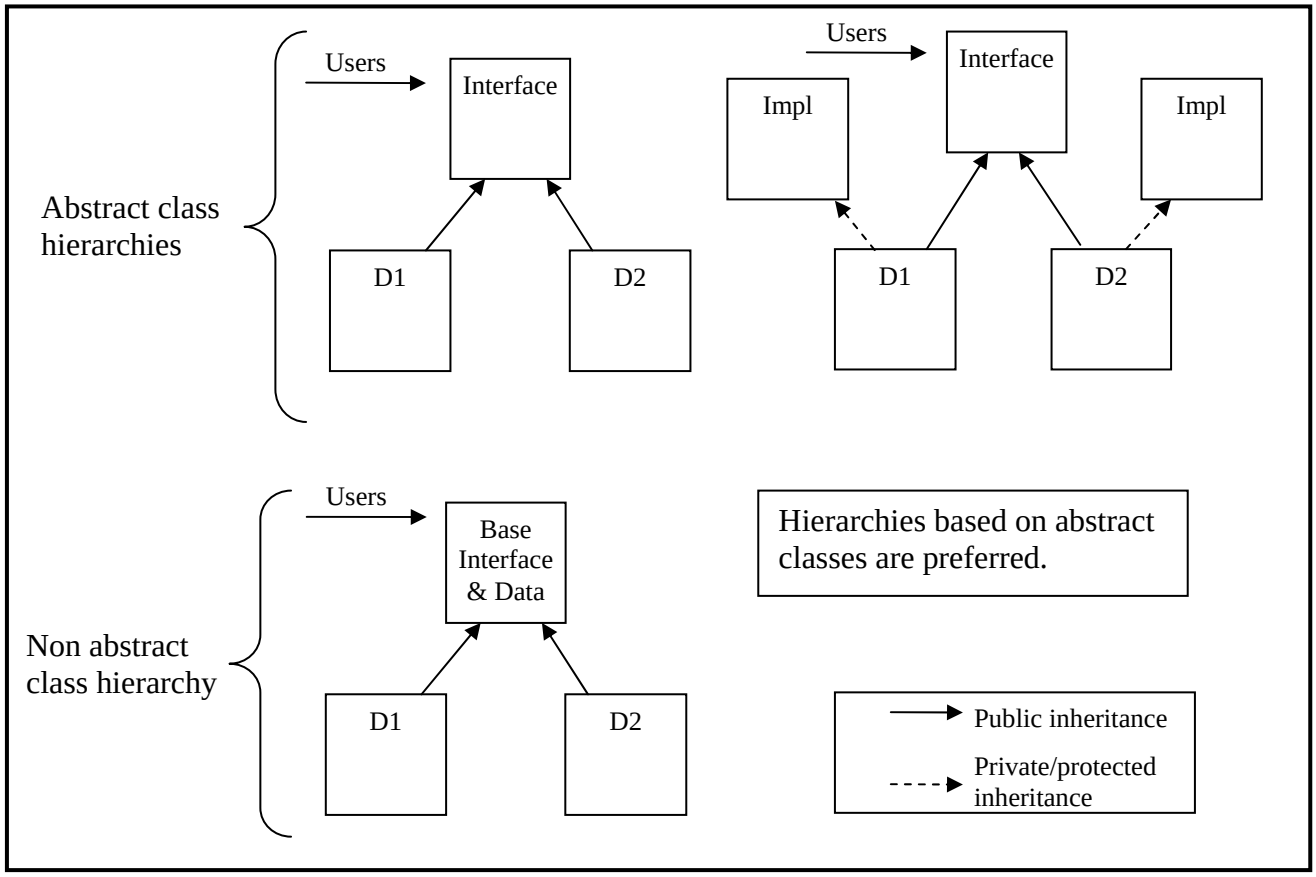
AV Rule 85

The following example illustrates how *operator!=()* may be defined in terms of *operator==()*. This construction simplifies maintenance.

```
bool  operator==(Sometype  a)
{
   if ( (a.attribute_1 == attribute_1) &&
        (a.attribute_2 == attribute_2) &&
        (a.attribute_3 == attribute_3) &&
        ...
        (a.attribute_n == attribute_n) )
   {
      return true;
   }
   else
   {
     return false;
   }

bool  operator!=(Some_type a)
{
   return !(*this==a);              //Note "!=" is defined in terms of "=="
}
```

AV Rule 87

Hierarchies based on abstract classes are preferred. Therefore the hierarchies at the top of the diagram are preferred over the hierarchy at the bottom of the diagram.



AV Rule 88

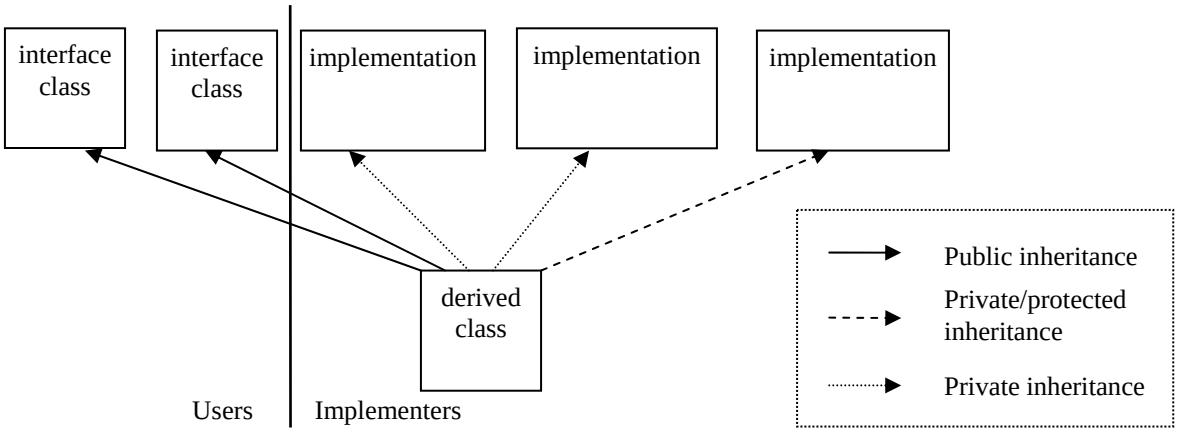
In the context of this rule, an interface is specified by a class which has the following properties:

- it is intended to be an interface,
- its public methods are pure virtual functions, and
- it does not hold any data, unless those data items are small and function as part of the interface (e.g. a unique object identifier).

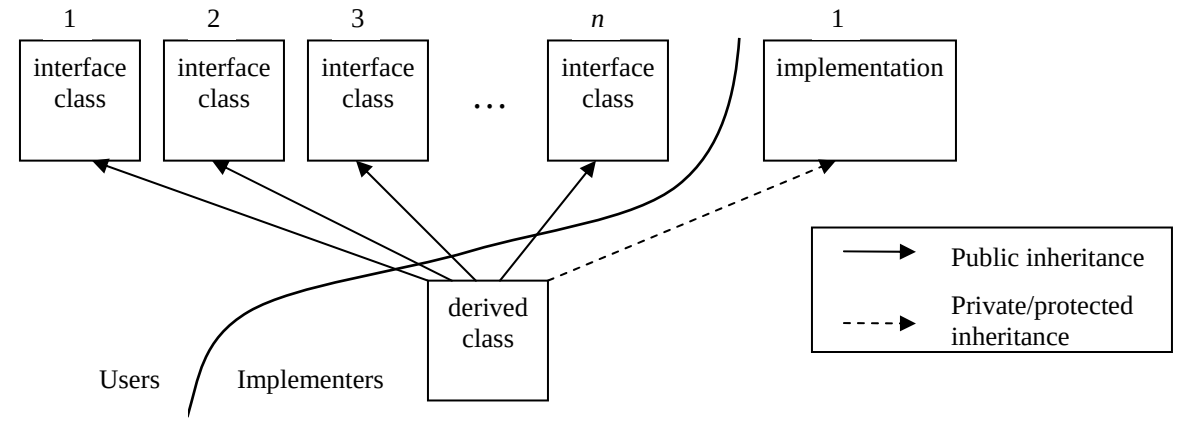
Note 1: Protected members may be used in a class as long as that class does not participate in a client interface.

Note 2: Classes with additional state information may be used as bases provided composition is performed in a disciplined manner with templates (e.g. policy-based design). See the "Programming with Policies" paragraph below.

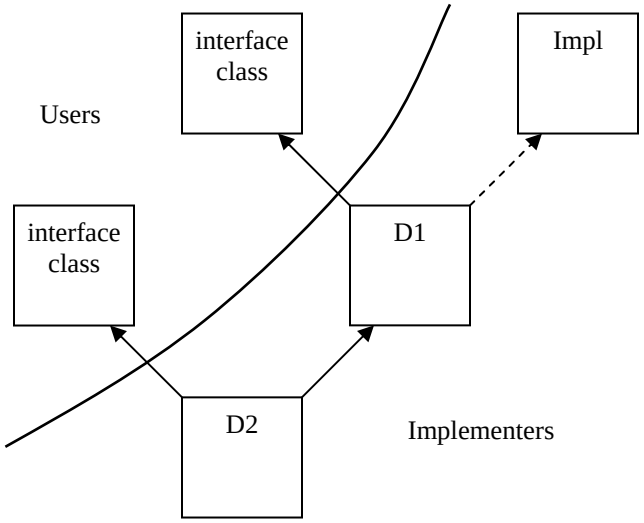
The following diagrams illustrate both good and bad examples of multiple inheritance.

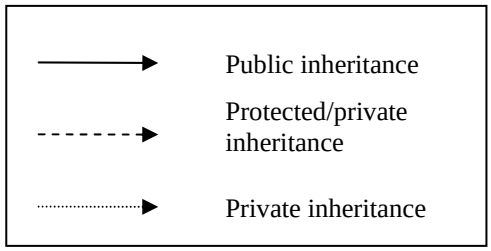


Good: Two interfaces, two private implementations, and one protected implementation.

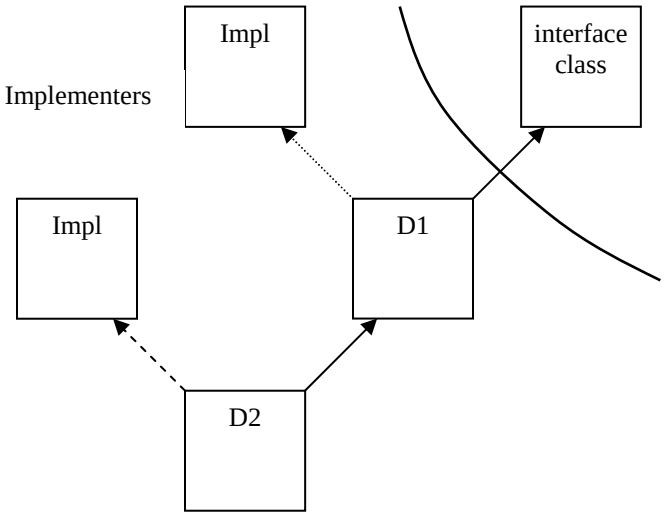


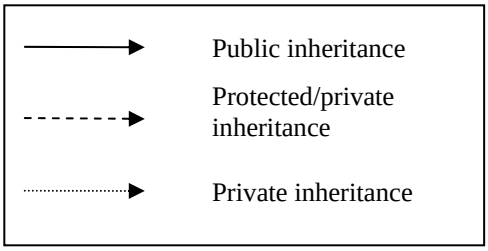
Good: *n* interfaces and one protected implementation.





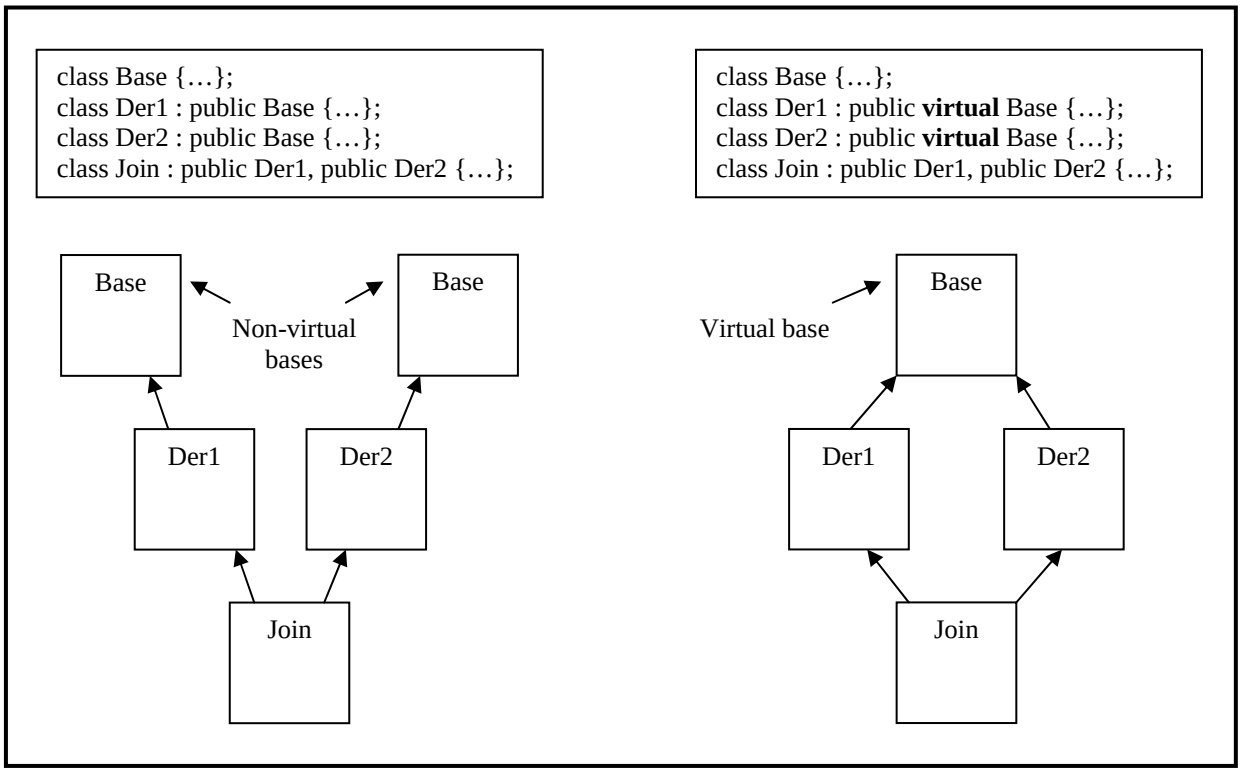
Good: D1 has one interface and one implementation.
D2 has two interfaces and one implementation.

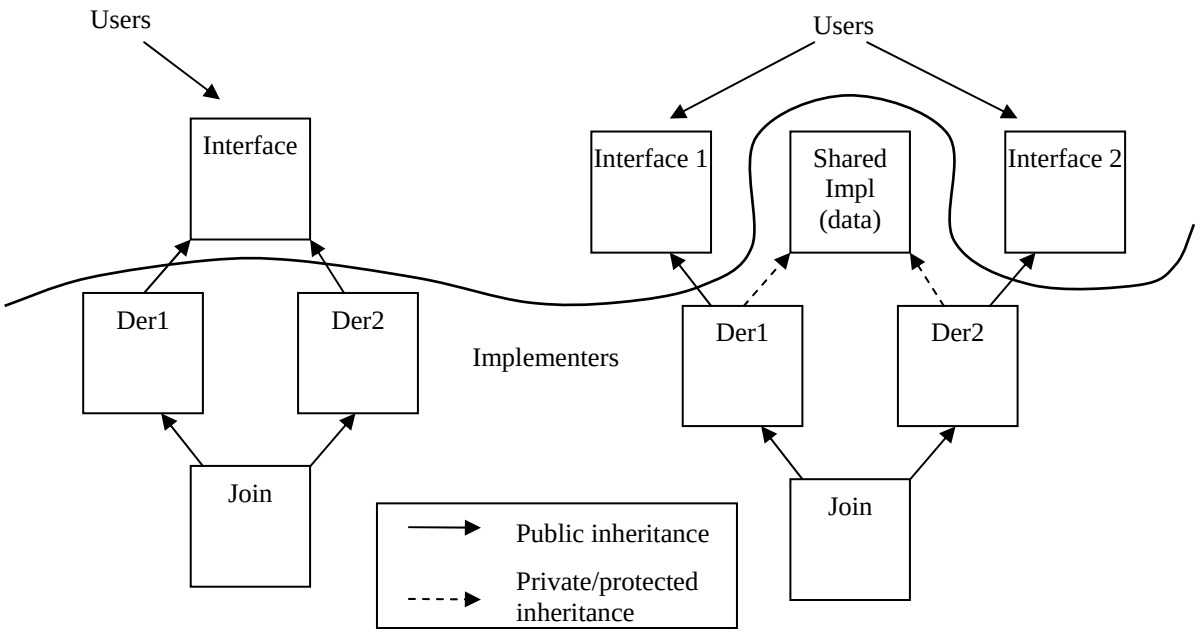




Good: Both D1 and D2 have one interface and one implementation.

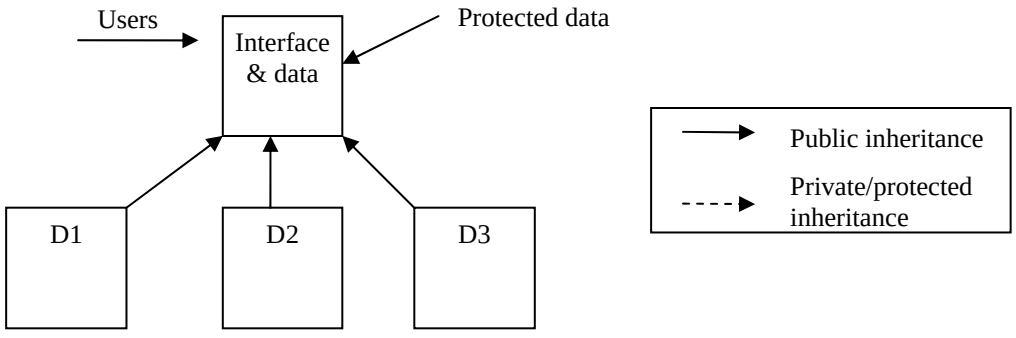
Virtual base classes: The following diagram illustrates the difference between virtual and non-virtual base classes. The subsequent diagram illustrates legitimate uses of virtual base classes.



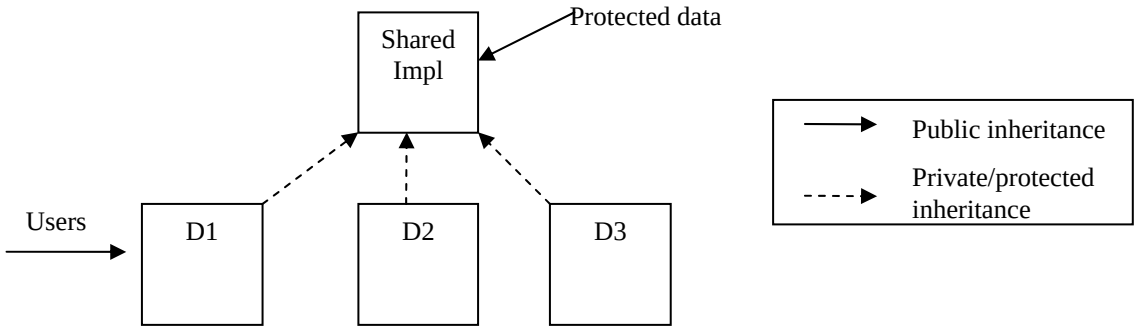


Good: Both hierarchies are acceptable. Note the Implementation class in the right hierarchy should not be directly used by clients.

Protected data in class interface: As previously mentioned, protected data may be used in a class as long as that class does not participate in a client interface. The following diagram illustrates this point.



Bad: Derived classes (D1, D2, or D3) may *hijack* the base class invariant since they have access to the base class protected data.



Good: The base class is an implementation class. Non-public inheritance prevents derived classes from being implicitly converted to the base class type. Hence users may only manipulate the derived classes through the derived class interfaces.

Policy-based Design

As previously mentioned, classes with additional state information may be used as bases provided composition is performed in a disciplined manner with templates (e.g. policy-based design). A form of programming that is used when classes must be customizable but efficiency is paramount is called policy programming. When a class' functionality can be separated into a set of independent concerns and each concern can be programmed in more than one way, policy programming is very useful. In particular, it simplifies maintenance by avoiding replication of code. A classic example is a matrix math package. The concerns are as follows:

- Access – how are the elements laid out in memory? Some possibilities are row major, column major, and upper triangular.
- Allocation – from where does the memory come? Some possibilities are the system heap, a fixed area, or allocated by a user-specified allocation scheme.
- Error Handling – what is done when an error occurs? Some possibilities are to throw an exception, log an error message, set an error code, or restart the process.

These concerns are independent of one another and can be coded separately. For example:

```
template< class T >
class Row_major
{
    public:
       typedef T value_type;
      Row_major( int32 nrows, int32 ncols, T* array ) :
          nrows_(nrows), ncols_(ncols), array_(array)
                                                    {}
      ~Row_major()                                  {}
       int32 size1() const                          { return nrows_; }
       int32 size2() const                          { return ncols_; }
       const T& operator() ( int32 i, int32 j ) const   { return array_[i*ncols_+j]; }
      T& operator() ( int32 i, int32 j )            { return array_[i*ncols_+j]; }

   private:
      int32 nrows_;
      int32 ncols_;
      T*  array_;
};
```

The class Column_major would be very similar except that the parenthesis operator would return array_*[j*nrows_+i]*.

Rather than create code for each possible combination of concerns, we create a template class that brings together implementations for each concern. Thus, assuming that:

- *Access* defines the parenthesis operator,
- *Alloc* defines the template method
  T allocate<T>(int32 n)* , and
- *Err* defines the following methods

void handle_error(int32 code, int32 nr, int32 nc)
void handle_error(int32 code, int32 i, int32 j, int32 nr, int32 nc)

we can compose the *Matrix* class as follows:

```
template< class Access, class Alloc, class Err >
class Matrix : public Access, Alloc, Err        // Alloc and Err are private bases
{
   Matrix( int32 nrows, int32 ncols ) :
         Access(nrows,ncols,allocate<T>(nrows*ncols))
   {
      if( array_==0 )
      {
         handle_error( Err::allocation_failed, nrows, ncols );
      }
   }

   Access::value_type& at( int32 i, int32 j )
   {
      if( i<0 || i>nrows_ || j<0 || j>ncols_ )
      {
         handle_error( Err::index_out_of_bounds, i, j, nrows_, ncols_ );
         i = j = 0;
      }
      return this->operator()(i,j);
   }

 // and so on...
};
```

Thus, the *Matrix* class brings all the policies together into a functional class. Users may create

- *Matrix< Row_major, Heap, Exceptions >* or
- *Matrix< Lower_triangular, Pool_allocation, Restart >*

as dictated by their needs.

Note that the *Matrix* class could have been written where *Access*, *Alloc*, and *Err* exist as data members of *Matrix* rather than deriving from it. This technique has several drawbacks including the necessity of creating (and maintaining) a large number of forwarding functions as well as inferior performance characteristics.

AV Rule 88.1

Stateful virtual bases should be rarely used and only after other design options have been carefully weighed. Stateful virtual bases do introduce a concern with respect to non-exclusive access to shared data. However, this concern is not unique to stateful virtual bases. On the contrary, it is present in any form of aliasing. For example, two pointers that point to a single data object suffer from the same condition, but this situation is arguably worse since there are no declarations in the system to highlight this form of aliasing (as there are for virtual bases).

Stateful virtual bases are theoretically important since they provide the only explicit means of sharing data within a class hierarchy without transitioning to a brittle, single-rooted hierarchy employing stateful bases. The other alternative is simpler and uglier yet: give each class that needs access to shared data a pointer to (1) a part of the object or to (2) a separate object - thus "simulating" a virtual base. In essence, a stateful virtual base should be used only to avoid the implicit sharing of data via pointers or references.

Consider the following hierarchy:

AV Rule 88.1 would make the fact that *A* is a virtual base explicit not only in the declarations of *B* and *C*, but also in the declarations of *D*, *E*, and *F* (assuming *D*, *E*, and *F* all access *A*):

```
struct A {};
struct B : virtual A {};
struct C : virtual A {};
struct D : B, virtual A {};
struct E : C, virtual A {};
struct F : D, E, virtual A {};
```

Consequently, the sharing of data is explicitly documented. The alternative:

```
struct A {};
struct B : virtual A {};
struct C : virtual A {};
struct D : B {};          // Violation of 88.1
struct E : C {};          // Violation of 88.1
struct F : D, E {};       // Violation of 88.1
```

can be obscure. That is, it is not obvious that D and E do not have exclusive access to A.

AV Rule 92

AV Rule 92 specifies that subtypes should conform to the Liskov Substitution Principle (LSP) which states:

> *…for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T [5].*

More simply put, the LSP suggests that a pointer or reference to a derived type may be substituted anywhere one of its base types is used without the context being aware of the substitution. Following this important principle will ensure that functions/modules can be constructed without requiring the context of a base class to be aware of all current and future derivatives of that base class. In other words, class hierarchies may be constructed so that new extensions/specializations will not break or yield surprise results when used in existing applications.

For example, should *Penguin* be derived from the base class, *Bird,* that contains the *fly()* operation? The precondition (*all birds can fly*) for the base class, *Bird,* is stronger than the precondition (*I can't fly*) of the derived class, *Penguin*. Hence, *Penguin* is not a subtype of *Bird,* and therefore should not be publicly derived from *Bird*.

AV Rule 93

Example A illustrates the class *Person* that is constructed with members *Name*, *Address,* and *PhoneNumber*. Hence, the functionality of *Person* is implemented in terms of the member elements (*Name*, *Address,* and *PhoneNumber*).

Example A:

```
class Person
{
  private:
    string      name;          // Person is composed of members Name, Address, and
                               // PhoneNumber
    string      address;
    string      phone_number;
…
};
```

In general, membership should be used except where access to protected members or virtual methods is required. In these situations, membership will not work. Instead, non-public inheritance should be used. Consider the *GenericStack* example provided by Meyers [6], item 43. One may reuse the *GenericStack* implementation for stacks of any type as illustrated in Example B. Note, however, that the *GenericStack* implementation is "too dangerous" to be used by it self. Instead, type-safe interfaces are supplied through a template class. The *GenericStack*'s methods are declared *protected* to prevent the use of this class in isolation from a type-safe interface. As a result, derived classes must make use of GenericStack's protected members via inheritance rather than class membership.

Example B:

```
class Generic_stack
{
 protected:                              // Methods are protected so that Generic_stack
                                         // cannot be used by itself.
    Generic_stack();
    ~Generic_stack();

     void        push   (void *object);
    void *       pop    (void);
    bool         empty () const;
 private:
…
};
```

A type-safe interface for *GenericStack* may be implemented as:

```
template<class T>

class Stack: private Generic_stack       // Reuse base class implementation
{
 public:
    void   push   (T *object_ptr)  { GenericStack::push (object_ptr); }
    T *    pop    (void)           { return static_cast<T*>(Generic_stack::pop()); }
    bool   empty () const          { return Generic_stack::empty(); }
};
```

AV Rule 94

Nonvirtual functions are statically bound. In essence, a nonvirtual function will *hide* its corresponding base class version. Hence a single derived class object may behave either as a base class object or as a derived class object depending on the way in which it was accessed—either through a base class pointer/reference or a derived class pointer/reference. To avoid this duality in behavior, nonvirtual functions should never be redefined.

Example:

```
class Base
{
  public:
    mf (void);
};

class Derived : public Base
{
 public:
   mf (void);
};
example_function(void)
{
 Derived        derived;
 Base*          base_ptr     = &derived;              // Points to derived
 Derived*       derived_ptr = &derived;               // Points to derived

 base_ptr->mf();          // Calls Base::mf()        *** Different behavior for same object!!
 derived_ptr->mf();       // Calls Derived::mf()
}
```

AV Rule 95

While C++ dynamically binds virtual methods, the default parameters of those methods are statically bound. Hence, the *draw()* method of the derived type *(Circle)*, if referenced through a base type pointer *(Shape *)*, will be invoked with the default parameters of the base type *(Shape)*.

Example A:

```
enum Shape_color { red, green, blue };
class Shape
{
  public:
    virtual void draw (Shape_color color = green) const;
    …
}
class Circle : public Shape
{
  public:
    virtual void draw (Shape_color  color = red) const;
    …
}
void fun()
{
  Shape* sp;

  sp = new Circle;
  sp->draw ();                  // Invokes Circle::draw(green) even though the default
}                               // parameter for Circle is red.
```

AV Rule 101 and AV Rule 102

Since many template instantiations may be generated, the compiler should be configured to provide a list of actual instantiations for review and testing purposes. The following table illustrates the output of a *Stack* class that was instantiated for both *float32* and *int32* types. Note that the method instantiations are listed so that a complete test plan may be constructed.

Template	Parameter Type	Library/Module
Stack<T1>::Stack<float32>(int)	[with T1=float32]	shape_hierarchy.a(shape_main.o)
Stack<T1>::Stack<int32>(int)	[with T1=int32]	shape_hierarchy.a(shape_main.o)
T1 Stack<T1>::pop()	[with T1=float32]	shape_hierarchy.a(shape_main.o)
T1 Stack<T1>::pop()	[with T1=int32]	shape_hierarchy.a(shape_main.o)
void Stack<T1>::push(T1)	[with T1=float32]	shape_hierarchy.a(shape_main.o)
void Stack<T1>::push(T1)	[with T1=int32]	shape_hierarchy.a(shape_main.o)

AV Rule 103

Stroustrup [4] provides a solution (for creating template parameter constraints) that requires minimal effort, requires no additional code to be generated, and causes compilers to produce acceptable error messages (including the word *constraint*).

Moreover, Stroustrup provides the following sample constraints that check the ability of template parameters to engage in derivations, assignments, comparisons and multiplications. (Note that the following elements are good candidates for a constraints library.)

```
template<class T, class B> struct Derived_from {
    static void constraints(T* p) { B* pb = p; }
    Derived_from() { void(*p)(T*) = constraints; }
};

template<class T1, class T2> struct Can_copy {
    static void constraints(T1 a, T2 b) { T2 c = a; b = a; }
    Can_copy() { void(*p)(T1,T2) = constraints; }
};

template<class T1, class T2 = T1> struct Can_compare {
    static void constraints(T1 a, T2 b) { a==b; a!=b; a<b; }
    Can_compare() { void(*p)(T1,T2) = constraints; }
};

template<class T1, class T2, class T3 = T1> struct Can_multiply {
    static void constraints(T1 a, T2 b, T3 c) { c = a*b; }
    Can_multiply() { void(*p)(T1,T2,T3) = constraints; }
};
```

Thus, given the *Can_copy* constraint above, a *draw_all()* function may be written that asserts, at compile time, that only containers comprised of pointers to *Shape* or pointers to a classes publicly derived from *Shape* (or convertible to *Shape*) may be passed in.

```
template<class Container>
void draw_all(Container& c)
{
   typedef typename Container::value_type T;
   Can_copy<T,Shape*>();                          // accept containers of only Shape*'s
   for_each(c.begin(),c.end(),mem_fun(&Shape::draw));
}
```

Additional constraints may be easily created. See [4] for further information concerning constraint creation and use.

AV Rule 108

The following example illustrates a case where function overloading or parameter defaults may be used instead of an unspecified number of arguments.

Example A: Consider a function to compute the length of two, three, or four dimensional vectors. A variable argument list could be used, but introduces unnecessary complexities. Alternatively, function overloading or parameter defaulting provide much better solutions.

// Unspecified number of arguments

```
float32  vector_length (float32  x, float32  y, …);          // Error prone
```

// Function overloading

```
float32  vector_length (float32 x, float32 y);
float32  vector_length (float32 x, float32 y, float32 z);
float32  vector_length (float32 x, float32 y, float32, z, float32 w);
```

// Default parameters

```
float32 vector_length (float32 x, float32 y, float32 z=0, float32 w=0);
```

AV Rule 109

In the following example, *Square* declares two functions *area()* and *morph()*. Since the designer wants to inline the relatively simple method *area()*, it is defined within the class specification. In contrast, there is no intent to inline the complex method *morph()*. Hence only the method declaration is included.

```
class Square : public Shape
{
  public:
    float32 area()
    {
      return length*width;
    }  // area() will be inlined since it is defined
       // in the class specification.

   morph (Shape &s);                    // morph() is not intended to be inlined so its
};                                      // implementation is contained in a separate file.
```

AV Rule 112

The following examples illustrate several ways in which function return values can obscure resource ownership and hence risk resource leakage. Note in the following examples, *new* need not allocate memory from the heap, but could be overloaded on the class in question.

Example A: Returning a dereferenced pointer initialized by new is error prone since the caller must remember to delete the object. This becomes more difficult if that object happens to be a temporary object.

```
X&   f (float32 a)
{
   return *new x(a);            // Error prone. Caller must remember to perform
}                               // the delete.

X&  ref = f(1);                 // The caller of f() must be responsible for deleting
…                               // the memory.
delete &ref                     // delete must be called for every invocation of f().
…
X&  x = f(1)*f(2)*f(3)*f(4);    // Memory leak: delete not called for temporaries.
```

Example B: Returning a pointer to a local object is problematic since the object ceases to exist after return. AV Rule 111 explicitly prohibited this practice.

```
X*  f (float32 a)           // Error: the caller most likely believes he is
{                           // responsible for deleting the object. However, the object
   X  b(a);                 // ceases to exist when the function returns.
   return &b
}
```

Example C: A function can return a pointer to an object, but the recipient must remember to perform the *delete*.

```
X *f(float32 a)
{
   return new X(a);     // Beware of leak: recipient must remember to perform the delete.
}
```

Example D: Returning an object by value is a simple method that does not obscure ownership issues.

```
X f(float 32 a)         // Simple and clear.
{
   X b(a);
   return b;
}
```

AV Rule 120

Overloading functions can be a powerful tool for creating families of related operations that differ only with respect to argument type. If not used consistently, however, overloading can lead to considerable confusion.

Example A: Proper usage of function overloading is illustrated below. All overloads of *contains()* share the same name as well as perform the same conceptual task.

```
class String
{
  public:                                        // Used like this:
    // ...                                       // String x = "abc123";
    int32 contains ( const char       c );      // int32 i   = x.contains( 'b' );
    int32 contains ( const char*      cs );     // int32 j   = x.contains( "bc1" );
    int32 contains ( const String&  s );        // int32 k   = x.contains( x );
    // ...
};
```

Example B: Improper use of operator overloading is illustrated below. For two-dimensional vectors, *operator*()* means *dot product* while for three dimensional vectors, *operator*()* means *cross product*.

```
Vector2d {
 public:
   float32  operator*(const Vector2d & v);        // compute dot product
   …
};
Vector3d {
 public:
   Vector3d  operator*(const Vector3d & v)       // compute cross product
   …
};
```

AV Rule 121

The Green Hills compiler employs two inlining approaches, each using a different inlining strategy, and each coming at a different stage. The first is a front-end inliner. It will only consider inline functions (functions declared with the keyword *inline* or member functions whose bodies are defined inside class definitions).

The front-end inliner will inline only those functions which can be converted to expressions. Therefore, functions which simply return an expression, straight code functions (which can be converted to comma expressions), or functions with *if* statements that can be converted to "?:" expressions will be considered candidates for inlining. The front-end inliner is not capable of inlining more complex statements (e.g. functions containing loops).

The second inliner is the independent code inliner which is capable of inlining most any function (except recursive functions). Inlining complex functions may lead to significant code bloat as well as to complicate debugging efforts. As a result, only the front-end inliner **should** be used in C/C++ programs.

AV Rule 122

The following example illustrates a class that inlines a trivial accessor and a trivial mutator.

```
class  Example_class
{
   public:
      int32 get_limit (void)                    // Sample accessor to be inlined
      {
         return limit;
       }
      void    set_limit (int32 limit_parm)      // Sample mutator to be inlined
      {
         limit = limit_parm;
       }
  private:
     int32  limit;
};
```

AV Rule 124

Simple forwarding functions should be inlined as illustrated below.

Example A:

```
inline draw()     // Example of a forwarding function that should be inlined
{
   draw_foreground ();
}
```

AV Rule 125

The construction of large or complex temporary objects can exact a significant performance penalty. Consequently, the following observations are provided as guidance in limiting the number unnecessary temporaries.

- **Problem 1:** Temporary objects are created (and destroyed) to make function calls succeed via implicit type conversions. The conversions will occur either when an argument is passed by value or is passed as a reference to const objects.
- **Solution 1:** Overload the function in question so that the implicit conversion will not be necessary.
- **Problem 2:** Temporary objects are created (and destroyed) when a function returns an object.
- **Solution 2a:** Return a reference when possible. If it is not possible to return a reference (as in the case of overloading *operator*()*), try to take advantage of "return value optimization" (eliminating a local temporary by utilizing the object at the functions return site). For example:

  ```
  c = a * b;
  …
  inline const Rational operator*(const Rational& lhs,
                                  const Rational& rhs)
  {
     return Rational (lhs.get_numerator()    * rhs.get_numerator(),
                      lhs.get_denominator() * rhs.get_denominator());
  }
  ```

  Eliminates both the temporary created inside the *operation*()* and the temporary returned by *operator*().* The new object is simply constructed inside the space allocated for "c".

- **Solution 2b:** Change the design. For example, use *operator*=()* instead of *operator*(),* since *operator*= ()* does not require the generation of a temporary as does *operator*().*

AV Rule 126

A C++ style comment begins with "//" and terminates with a new-line. However, the placement of vertical-tab or form-feed characters within a comment may produce unexpected results. That is, if a form-feed or a vertical-tab character occurs in a C++ style comment, only white-space characters may appear between it and the new-line that terminates the comment. An implementation is not required to diagnose a violation of this rule. [10]

AV Rule 136

The following code illustrates some problems encountered when a variable is not declared at the smallest feasible scope.

```
void fun_1()
{
   int32 i;                        // Bad: i is prematurely declared (the intent is to use i in the
                                   //       for loop only)
   …                               // Bad: i has a meaningless value in the region of the code
   for (i=0 ; i<max ; ++i)
   {
      …
   }
   ….                              // Bad: i should not be used here, but could be used anyway

   for(int32 j=0 ; j<max ; ++j)    // Good: j is not declared or initialized until needed
   {                               // Good: j is only known within the for loop's scope
      …
   }
}
```

AV Rule 137

MISRA Reason: Declarations at file scope are external by default. Therefore if two files both declare an identifier with the same name at file scope, the linker will either give an error, or they will be the same variable, which may not be what the programmer intended. This is also true if one of the variables is in a library somewhere. Use of the *static* storage-class specifier will ensure that identifiers are only visible to the file in which they are declared.

If a variable is only to be used by functions within the same file then use ***static***. Similarly if a function is only called from elsewhere within the same file, use ***static***.

Typically, functions whose declarations appear in a header (.h) file are intended to be called from other files and should therefore never be specified with the ***static*** keyword. Conversely, functions whose declarations appear in an implementation body (.cpp) file should never be called from other files, and hence should always be declared with the ***static*** keyword.

AV Rule 138

The C++ Standard [10] defines linkage in the following way:

- When a name has external linkage, the entity it denotes can be referred to by names from scopes of other translation units or from other scopes of the same translation unit.
- When a name has internal linkage, the entity it denotes can be referred to by names from other scopes in the same translation unit.

Hence, having names with both internal and external linkage can be confusing since the objects to which they actually refer may not be obvious. Consider the following example where the *i* declared on line 1 has internal linkage while the *i* on line 2 has external linkage. Which entity is referenced by the *i* on line 3?

```
{
   static int32 i=1;          // line 1
   {                          // Bad: the i with external linkage hides the i
                              //      with internal linkage.
      extern int32 i;         // line 2
      …
      a[i] = 10;              // line 3: Confusing: which i?
   }
}
```

AV Rule 139

Adherence to this rule will normally mean declaring external objects in header files which will then be included in all those files that use those objects (including the files which define the objects).

Example A: Two files declare the same variable. This style could lead to errors since *a* could be declared in many different files. A change in one of those files would affect all others and would be difficult to pinpoint.

```
// In File_1.cpp
int32   a = 3;

// In File_2.cpp
extern int32   a;
```

Example B: Here, *a* is declared in a header file. All other files that need access to *a* simply include the header file. In this way, consistency is assured.

```
// In File_1.h
extern int32 a;

// In File_1.cpp
#include <File_1.h>
int32   a = 3;

// In File_2.cpp
#include <File_1.h>
```

AV Rule 141

Example A: Declaring an enumeration in the definition of its type can lead to readability problems and unnamed data types as illustrated below.

```
enum                    // Don't do this: Creates an unnamed data type.
{
   up,
   down
} direction;
enum i { in, out } i;   // Don't do this: Difficult to read.
```

Example B: Separation of the declaration and definition are preferred as illustrated below. Note that this requires the data type to be named which provides a mechanism to create other variables of the same type and the ability to type cast.

```
enum XYZ_direction
{
   up,
   down
};
XYZ_direction   direction;
```

Example C: Note that a legitimate use of an unnamed enumeration is to define symbolic constants within a class declaration.

```
class X
{
   enum
   {
     max_length = 100,
     max_time   =   73
   };                   // Defines symbolic constants for the class
   …
};
```

Example D: Note that the following declarations are not prohibited under this rule.

```
int32 i=0;
pair<float32,int32> p;
```

AV Rule 142

MISRA Rule 30 requires all automatic variables to have an assigned value. Compilers will, by default, initialize external and static variables to the value zero. However, it is considered good practice to initialize all variables, not just automatic/stack variables, to an initial value for purposes of 1) clarity and 2) bringing focused attention to the initialization of each variable. Therefore, this rule requires ALL variables to be initialized. Exception may be granted for volatile variables.

AV Rule 143

Introducing variables before they can be assigned meaningful values causes a number of problems as illustrated in the following examples.

Example A: The following code illustrates some problems encountered when variables are introduced before they can be properly initialized.

```
void fun_1()                    // Poor implementation
{
   int32  i;                    // Bad: i is prematurely declared (the intent is to use i in the for
                                //      loop only)
   int32  max=0;                // Bad: max initialized with a dummy value.
   …                            // Bad: i and max have meaningless values in this
                                //      region of the code.
   max = f(x);
   for (i=0 ; i<max ; ++i)
   {
      …
   }
   ….                           // Bad: i should not be used here, but could be used anyway
}

void fun_1()                            // Good implementation
{
   ….
   int32 max = f(x);                    // Good: max not introduced until meaningful value is
                                        //       available
   for (int32 i=0 ; i<max ; ++i)        // Good: i is not declared or initialized until needed
   {                                    // Good: i is only known within the for loop's scope
      …
   }
}
```

Example B: An instance of class *X* is constructed prior to the point at which it can be fully initialized. To complete the initialization, a separate *init()* method must be called when sufficient information becomes available. However, since the object may only be in a quasi-valid state prior to the invocation of *init(),* all method invocations between object construction and *init()* are suspect. See also AV Rule 73 concerning unnecessary default constructors.

```
class X {
  public:
    X::X() {}                  // Bad: default constructor builds partially initialized object.
    init (int32 max_, int32 min_)
    {
       max = _max ;
       min = _min;
    }
    int32 range()
    {
       return max-min ;
    }
    …
  private:
    int32 max;
    int32 min;
};

void foo()
{
    X x;                              // Bad: x constructed but without data
    …
    x.range();                        // Bad: undefined result.
   ….
    x.init(lbound, ubound);           // Bad: x initialized later than necessary
}
```

AV Rule 145

If an enumerator list is given with no explicit initialization of members, then C++ allocates a sequence of integers starting at 0 for the first element and increasing by 1 for each subsequent element. For most purposes this will be adequate.

An explicit initialization of the first element, as permitted by the above rule, forces the allocation of integers to start at the given value. When adopting this approach it is essential to ensure that the initialization value used is small enough that no subsequent value in the list will exceed the *int* storage used by enumeration constants.

Explicit initialization of all items in the list, which is also permissible, prevents the mixing of automatic and manual allocation, which is error prone. However it is then the responsibility of the programmer to ensure that all values are in the required range, and that values are not unintentionally duplicated.

Example A:

```
//Legal enumerated list using compiler-assigned enum values
//off=0, green=1, yellow=2, red=3

enum Signal_light_states_type
{
  off,
  green,
  yellow,
  red
};
```

Example 2:

```
// Legal enumeration, assigning a value to the first item in the list.

enum Channel_assigned_type
{
  channel_unassigned = -1,
  channel_a,
  channel_b,
  channel_c
};
```

Example 3:

```
// Control mask enumerated list.  All items explicitly
// initialized.

enum FSM_a_to_d_control_enum_type
{
  start_conversion        = 0x01,
  stop_conversion         = 0x02,
  start_list              = 0x04,
  end_list                = 0x08,
  reserved_3_bit          = 0x70,
  reset_device            = 0x80
};
```

Example 4:

```
// Legal: standard convention used for enumerations that are intended to index arrays.

enum Color {
  red,
  orange,
  yellow,
  green,
  blue,
  indigo,
  violet,
  Color_begin = red,
  Color_end = violet,
  Color_NOE                    // Number of elements in array
};
```

AV Rule 147

Manipulating the underlying bit representation of a floating point number is error-prone, as representations may vary from compiler to compiler, and platform to platform. There are, however, specific built-in operators and functions that may be used to extract the mantissa and exponent of floating point values.

AV Rule 151.1

Since string literals are constant, they should only be assigned to constant pointers as indicated below:

```
char* c1 = “Hello”;          // Bad: assigned to non-const
char c2[] = “Hello”;         // Bad: assigned to non-const
char c3[6] = “Hello”;        // Bad: assigned to non-const
c1[3] = ‘a’;                 // Undefined (but compiles)

const char* c1 = “Hello”;    // Good
const char c2[] = “Hello”;   // Good
const char c3[6] = “Hello”;  // Good
c1[3] = ‘a’;                 // Compile error
```

AV Rule 157

Care should be taken when short-circuit operators are utilized. For example, if the logical expression in the following code evaluates to *false*, the variable *x* will not be incremented. This could be problematic since subsequent statements may assume that *x* has been incremented.

```
if  ( logical_expression && ++x)     // Bad: right-hand side not evaluated if the logical
                                     //       expression is false.
…
f(x);                                // Error: Assumes x is always incremented.
…
```

AV Rule 158

The intent of this rule is to require parenthesis where clarity will be enhanced while stopping short of over-parenthesizing expressions. In the following examples, parenthesizing operands (that contain binary operators) of the logical operators && or || enhances readability.

Examples:

```
valid (p)     &&   add(p)          // parenthesis not required
x.flag        &&   y.flag          // parenthesis not required
a[i]           ||   b[j]           // parenthesis not required

(x < max )  &&   (x > min)         // parenthesis required
(a || b)      &&   (c || d)        // parenthesis required
```

AV Rule 160

The intent of this rule is to prohibit assignments in contexts that are obscure or otherwise easily misunderstood. The following example illustrates some of the problems this rule addresses.

Note that a *for-init-statement* (that is not a declaration) is an expression statement.

`for` (*for-init-statement condition-opt* ; *expression-opt*) *statement*

for-init-statement:
expression-statement
simple-declaration

Examples:

```
x = y;                  // Good: the intent to assign y to x and then check if x is
if (x != 0)             //       not zero is explicitly stated.
{
   foo ();
}
if ( ( x = y) != 0 )    // Bad: not as readable as it could be.
{                       //      Assignment should be performed prior to the "if"statement
  foo ();
}
if (x = y)              // Bad: intent is very obscure: a code reviewer could easily
{                       //      think that "==" was intended instead of "=".
   foo ();
}

for (i=0 ; i<max ; ++i) // Good: assignment in expression statement of "for" statement
{
   …
}
```

AV Rule 168

MISRA Rule 42 only allows use of the comma operator in the control expression of a *for* loop. The comma operator can be used to create confusing expressions. It can be used to exchange the values of variable array elements where the exchange appears to be a single operation. This simplicity of operation makes the code less intuitive and less readable. The comma operator may also be easily confused with a semicolon used in the *for* loop syntax. Therefore, all uses of the comma operator will not be allowed.

AV Rule 177

User-defined conversion functions come in two forms: single-argument constructors and type conversion operators. Implicit type conversions may be eliminated as follows:

- Single-argument constructors: use the "explicit" keyword on single-argument constructors so that the compiler will not supply implicit conversions through the constructor.
- Type conversion operators: don't define conversion operators. If type conversion functionality is required, then define a member function to fulfill the same role. Unlike the type conversion operator, however, a member function must be called explicitly, thus eliminating any "surprises" that could arise if the type conversion operator were used.

Examples 1 and 2 demonstrate these principles.

Example 1a: The *Vector_int* class below has a single argument constructor used to build vectors. However, this constructor may be called in ways a user may not expect. The solution is to use the *explicit* keyword in the constructor declaration which precludes the constructor from being called implicitly.

```
bool operator == (const Vector_int &lhs, const Vector_int &rhs)
{
  // compare two Vector_ints
}
class Vector_int {
  public:
    Vector_int (int32 n);
    …
};

Vector_int   v1(10),
             v2(10);          // create two vectors of size 10;
…
for (int32 i=0 ; i<10 ; ++i)
{
   if (v1 == v2[i])           // The programmer meant to compare the elements of two Vectors.
   {                          // However, the subscript of the first was inadvertently left off.
   …                          // Thus, the compiler is asked to compare a Vector_int with an
   }                          // integer. The single argument constructor is called to convert the
}                             // integer to a new Vector_int so that the comparison can take place.
                              // This is almost certainly not what is expected.
```

Example 1b: The constructor is declared explicit so that the error is caught at compile time.

```
class Vector_int
{
  public:
    explicit Vector_int (int32 n) ;
  …
};

Vector_int    v1(10), v2(10);       // create two vectors of size 10;
…
for (int32 i=0 ; i<10 ; ++i)
{
    if (v1 == v2[i])            // The programmer meant to compare the elements of two Vectors.
    {                           // However, the subscript of the first was inadvertently left off.
    …                           // Thus, the compiler is asked to compare a Vector_int with an
    }                           // integer. The explicit keyword prevents the constructor from
}                               // being called implicitly, so the compiler generates an error.
```

Example 2a: Class *Complex* defines a complex number, but the output operator has not been defined for the class. Thus, when the user attempts to print out a complex number, an error is not generated. Instead, the number is *silently* converted to a real number by the conversion operator. This yields a potentially surprising result to the client.

```
class Complex
{
  public:
    Complex (double r, double i = 1) : real(r), imaginary(i) {}     // Constructor
    operator double() const;                                        // Conversion operator
    …                                                               // converts Complex to double
  private:
    double real;
    double imaginary;
};

Complex r(1,2);

cout << r << endl;                  // User might expect compile error, but instead
                                    // r is automatically converted to decimal form
                                    // potentially losing information.
```

Example 2b: Instead of the conversion operator, class *Complex* now has a member function that performs the same role. Hence, the same functionality is maintained but without any potential surprises.

```
class Complex
{
  public:
    Complex (double r, double i = 1) : real(r), imaginary(i) {}     // Constructor
    double  as_double() const;                                      // Conversion operator
    …                                                               // converts Complex to double
  private:
    double real;
    double imaginary;
};

Complex r(1,2);

cout << r << endl;                                                  // Compile error generated.
cout << r.asDouble() << endl;                                       // Called explicitly rather than
                                                                    // implicitly.
```

AV Rule 180

The following examples illustrate implicit conversions that result in the loss of information:

```
int32       i =1024;
char        c = i;          // Bad: (integer-to-char) implicit loss of information.

float32     f = 7.3;
int32       j= f;           // Bad: (float-to-int) implicit loss of information.

int32       k = 1234567890;
float32     g = k;          // Bad: (int-to-float) implicit loss of information
                            //      (g will be 1234567936)
```

Note that an explicit cast to a narrower type (where the loss of information could occur) may be used only where specifically required algorithmically. The explicit cast draws attention to the fact that information loss is possible and that appropriate mitigations must be in place.

AV Rule 185

Traditional C-style casts raise several concerns. First, they enable most any type to be converted to most any other type without any indication of the reason for the conversion

Next, the C-style cast syntax:

(type) expression *// convert expression to be of type **type**.*

is difficult to identify for both reviewers and tools. Consequently, both the location of conversion expressions as well as the subsequent analysis of the conversion rationale proves difficult for C-style casts.

Thus, C++ introduces several new-style casts (const_cast, dynamic_cast[4], reinterpret_cast, and static_cast) that address these problems. The new-style casts have the following form:

```
const_cast<type>        (expression)    // convert expression to be of type type.
reinterpret_cast<type>  (expression)
static_cast<type>       (expression)
```

Not only are these casts easy to identify, but they also communicate more precisely the developer's intent for applying a cast.

See also rule AV Rule 178 concerning conversions between derived classes and base classes.

AV Rule 187

ISO/IEC 14882 defines a side effect as a change in the state of the execution environment. More precisely,

> *Accessing an object designated by a volatile lvalue, modifying an object, calling a library I/O function, or calling a function that does any of those operations are all side effects, which are changes in the state of the execution environment.*

Example: Potential side effect

```
if (flag)          // Has side effect only if flag is true.
{
  foo();
}
```

Example: The following expression has **no** side effects

- *3 + 4;* *// Bad: statement has zero side effects*

Example: The following expressions have side effects

- *x = 3 + 4;* *// Statement has one side effect: x is set to 7.*
- *y = x++;* *// Statement two side effects: y is set to x and x is incremented.*

[4] Note that dynamic casts are not allowed at this point due to lack of tool support, but could be considered at some point in the future after appropriate investigation has been performed for SEAL1/2 software.

AV Rule 192

Providing a final *else* clause (or comment indicating why a final *else* clause is unnecessary) ensures all cases are handled in an *else if* series as illustrated by the following examples.

Example A: Final *else* clause not needed since there is no *else if*.

```
if (a < b)
{
   foo();
}
```

Example B: Final *else* clause needed in case none of the prior conditions are satisfied.

```
if (a < b)
{
   …
}
else if (b < c)
{
   …
}
else if (c < d)
{
}
else                  // Final else clause needed
{
   handle_error();
}
```

Example C: Final *else* clause not needed, since all possible conditions are handled. Therefore a comment is included to clarify this condition.

```
if (status == error1)
{
   handle_error1();
}
else if (status == error2)
{
   handle_error2()
}
else if (status == error3)
{
    handle_error3()
}                              // No final else needed: all possible errors are accounted for.
```

AV Rule 193

Terminating a non-empty *case* clause in a *switch* statement with a *break* statement eliminates potentially confusing behavior by prohibiting control from *falling through* to subsequent statements. In the example below, primary and secondary colors are handled similarly so break statements are unneeded within each group. However, every non-empty case clause must be terminated with a break statement for this segment of code to work as intended.

Note: Omitting the final *default* clause allows the compiler to provide a warning if all enumeration values are not tested in the switch statement.

```
switch (value)
{
   case red :                                  // empty since primary_color() should be called
   case green :                                // empty since primary_color() should be called
   case blue :    primary_color (value);
                  break;                       // Must break to end primary color handling
   case cyan :
   case magenta :
   case yellow :  secondary_color (value);
                  break;                       // Must break to end secondary color handling
   case black :   black (value);
                  break;
   case white :   white (value);
                  break;
}
```

AV Rule 204

AV Rule 204 attempts to prohibit side-effects in expressions that would be unclear, misleading, obscure, or would otherwise result in unspecified or undefined behavior. Consequently, an operation with side-effects will only be used in the following contexts:

Note: It is permissible for a side-effect to occur in conjunction with a constant expression. However, care should be taken so that additional side-effects are not "hidden" within the expression.

Note: Functions *f()*, *g()*, and *h()* have side-effects.

1. by itself

```
++i;                              // Good
for (int32 i=0 ; i<max ; ++i)     // Good: includes the expression portion of a
                                  //       for statement

i++ - ++j;                        // Bad: operation with side-effect doesn't occur by itself.
```

2. the right-hand side of an assignment

```
y = f(x);                         // Good
y = ++x;                          // Good: logically the same as y=f(x)
y = (-b + sqrt(b*b -4*a*c))/(2*a); // Good: sqrt() does not have side-effect
y = f(x) + 1;                     // Good: side-effect may occur with a constant

y = g(x) + h(z);                  // Bad: operation with side-effect doesn't occur by itself
                                  //      on rhs of assignment
k = i++ - ++j;                    // Bad: same as above
y = f(x) + z;                     // Bad: same as above
```

3. a condition

```
if (x.f(y))                       // Good
if (int x = f(y))                 // Good: this form is often employed with dynamic casts
                                  //       if (D* pd = dynamic_cast<D*> (pb)) {…}
if (++p == NULL)                  /// Good: side-effect may occur with a constant

if (i++ - --j)                    // Bad: operation with side-effect doesn't occur by itself
                                  //      in a condition
```

4. the only argument expression with a side-effect in a function call

```
f(g(z));                          // Good
f(g(z),h(w));                     // Bad: two argument expressions with side-effects
f(++i,++j);                       // Bad: same as above

f(g(z), 3);                       // Good: side-effect may occur with a constant
```

5. condition of a loop

```
while (f(x))                    // Good
while(--x)                      // Good

while((c=*p++) != -1)           // Bad: operation with side-effect doesn't occur by itself
                                //      in a loop condition
```

6. switch condition

```
switch (f(x))                   // Good

switch (c = *p++)               // Bad: operation with side-effect doesn't occur by itself
                                //      in a switch condition
```

7. single part of a chained operation

```
x.f().g().h();                  // Good
a + b * c;                      // Good: (operator+() and operator*() are overloaded)
cout << x << y;                 // Good
```

AV Rule 204.1

Since the order in which operators and subexpression are evaluated is unspecified, expressions must be written in a manner that produces the same value under any order the standard permits.

```
i = v[i++];              // Bad: unspecified behavior
i = ++i + 1;             // Bad: unspecified behavior
p->mem_func(*p++);       // Bad: unspecified behavior
```

AV Rule 207

Unencapsulated global data can be dangerous and thus should be avoided. Note that objects with only *get* and *set* methods, or *get* and *set* methods for each attribute are not considered to be encapsulated.

```
int32 x=0;               // Bad: Unencapsulated global object.

class Y {
   in32 x;
   public:
      Y(int32 y_);
      int32 get_x();
      void set_x();
};

Y y (0);                 // Bad: Unencapsulated global object.
```

AV Rule 209

A *UniversalTypes* file will be created to define all standard types for developers to use. The types include:

```
bool,                            // built-in type
char,                            // built-in type
int8, int16, int32, int64,       // user-defined types
uint8, uint16, uint32, uint64,   // user-defined types
float32, float64                 // user-defined types
```

Note: Whether *char* represents signed or unsigned values is implementation-defined. However, since modern implementations almost exclusively treat *char* as *unsigned char*, the built-in *char* type will be used under the assumption that it is unsigned.

AV Rule 210.1

This rule is intended to prohibit an application from making assumptions concerning the order in which non-static data members, separated by an access specifier, are ordered.

Consider **Example A** below. Class ***A*** can not be reliably "overlayed" on incoming message data, since attribute ordering (across access specifiers) is unspecified.

In contrast, structure ***B*** may be reliably "overlayed" on the same incoming message data.

Example A:

```
class A
{
…
        protected:      // a could be stored before b, or vice versa
                int32 a;
        private:
                int32 b;
};
…
                        // Bad: application assumes that objects of
                        //      type A will always have attribute a
                        //      stored before attribute b.
A* a_ptr = static_cast<A*>(message_buffer_ptr);
```

Example B:

```
struct B
{
   int32 a;
   int32 b;
};
…
        // Good: attributes in B not separated
        //          by an access specifier
B* b_ptr = static_cast<B*>(message_buffer_ptr);
```

AV Rule 213

Parentheses should be used to clarify operator precedence rules to enhance readability and reduce mistakes. However, overuse of parentheses can *clutter* an expression thereby reducing readability. Requiring parenthesis below arithmetic operators strikes a reasonable balance between readability and clutter.
Table 2 documents C++ operator precedence rules where items higher in the table have precedence over those lower in the table.

Examples: Consider the following examples. Note that parentheses are required to specify operator ordering for those operators below the arithmetic operators.

```
x = a * b + c;              // Good: can assume "*" binds before "+"
x = v->a + v->b + w.c;      // Good: can assume "->" and "." Bind before "+"

x = (f()) + ((g()) * (h()));  // Bad: overuse of parentheses. Can assume
                              //      function call binds before "+" and "*"
x = a & b | c;              // Bad: must use parenthesis to clarify order
x = a >> 1 + b;             // Bad: must use parenthesis to clarify order
```

Table 2 Operator Precedence [2]

Operator	Description	Associativity
scope resolution scope resolution global global	class_name :: member namespace_name :: member :: name :: qualified-name	left-to-right left-to-right right-to-left right-to-left
member selection member selection subscripting function call value construction post increment post decrement type identification run-time type identification run-time checked conversion compile-time checked conversion unchecked conversion *const* conversion	object . member pointer -> member pointer [expr] expr (expr_list) type (expr_list) lvalue ++ lvalue – **typeid** (type) **typeid** (expr) **dynamic_cast** < type > (expr) **static_cast** < type > (expr) **reinterpret_cast** < type > (expr) **const_cast** < type > (expr)	left-to-right
size of object size of type pre increment pre decrement complement not unary minus unary plus address of dereference create (allocate) create (allocate and initialize) create (place) create (place and initialize) destroy (deallocate) destroy array cast (type conversion)	**sizeof** expr **sizeof** (type) ++ lvalue -- lvalue ~ expr ! expr - expr + expr & lvalue * expr **new** type **new** type (expr-list) **new** (expr-list) type **new** (expr-list) type (expr-list) **delete** pointer **delete** [] pointer (type) expr	right-to-left
member selection member selection	object .* pointer-to-member pointer ->* pointer-to-member	left-to-right
multiply divide modulo (remainder)	expr * expr expr / expr expr % expr	left-to-right
add (plus) subtract (minus)	expr + expr expr – expr	left-to-right
shift left	expr << expr	left-to-right

shift right	expr >> expr	
less than less than or equal greater than greater than or equal	expr < expr expr <= expr expr > expr expr >= expr	left-to-right
equal not equal	expr == expr expr != expr	left-to-right
bitwise AND	expr & expr	left-to-right
bitwise exclusive OR	expr ^ expr	left-to-right
bitwise inclusive OR	expr | expr	left-to-right
logical AND	expr && expr	left-to-right
logical OR	expr || expr	left-to-right
conditional expression	expr ? expr : expr	right-to-left
simple assignment multiply and assign divide and assign modula and assign add and assign subtract and assign shift left and assign shift right and assign AND and assign inclusive OR and assign exclusive OR and assign	lvalue = expr lvalue *= expr lvalue /= expr lvalue %= expr lvalue += expr lvalue -= expr lvalue <<= expr lvalue >>= expr lvalue &= expr lvalue |= expr lvalue ^= expr	right-to-left
throw exception	**throw** expr	right-to-left
comma (sequencing)	expr , expr	left-to-right

AV Rule 214

The order of initialization for non-local static objects may present problems. For example, a non-local static object may not be used in a constructor if that object will not be initialized before the constructor runs. At present, the order of initialization for non-local static objects, which are defined in different compilation units, is not defined. This can lead to errors that are difficult to locate.

The problem may be resolved by moving each non-local static object to its own function where it becomes a local static object. If the function returns a reference to the local static object it contains, then clients may access the object via the function without any of the initialization order problems. Note that the function can be inlined to eliminate the function call overhead.

Example:

```
// file 1
static  int32  x = 5;

// file 2
static  int32  y = x + 1;          // Bad assumption.  The compiler might not have initialized
                                   // static variable x.
```

The solution is to substitute static local objects for static non-local objects since the creation time is precisely defined for static local objects: the first time through the enclosing function.

```
inline Resource& the_resource()
{
    static Resource  r;
    return r;
}
```

Now clients may at any time reference the *Resource* object, *r*, as *the_resource()* without consideration for the order of initialization among *r* and any other similarly defined local static objects.

Alternately, one might consider allocating objects in a memory pool or on a stack at startup.

AV Rule 215

Pointers should be eliminated from user interfaces wherever possible. Instead, objects with well defined interfaces should be used to *hide* pointers from clients as well as to ensure any pointer manipulation would be performed in a well-defined manner. For example, passing an ***Array*** object (instead of a raw array) through an interface eliminates the *array decay* problem and hence any pointer arithmetic required on the receiving end.

AV Rule 216

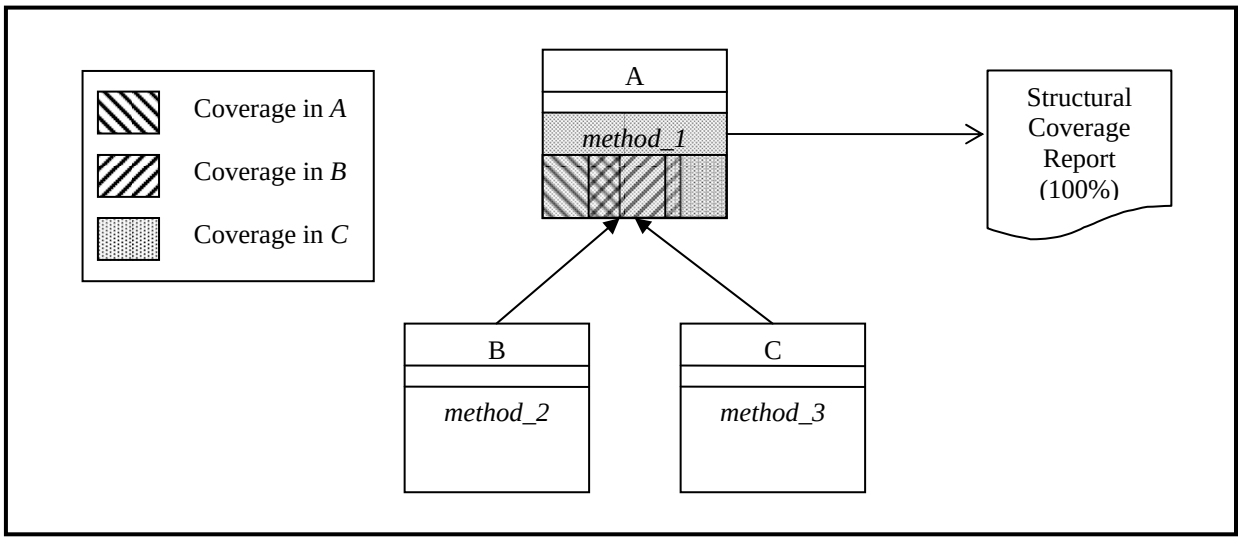
The overall performance of a program is usually determined by a relatively small portion of code. This is often referred to as the “80-20 Rule” which states that 80% of the time is spent in only 20% of the code. Thus, design and coding decisions should be made from a safety and clarity perspective with efficiency as a secondary goal. Only after adequate profiling analysis has been performed (where the true bottlenecks have been identified) should attempts at optimization be made.

AV Rule 220

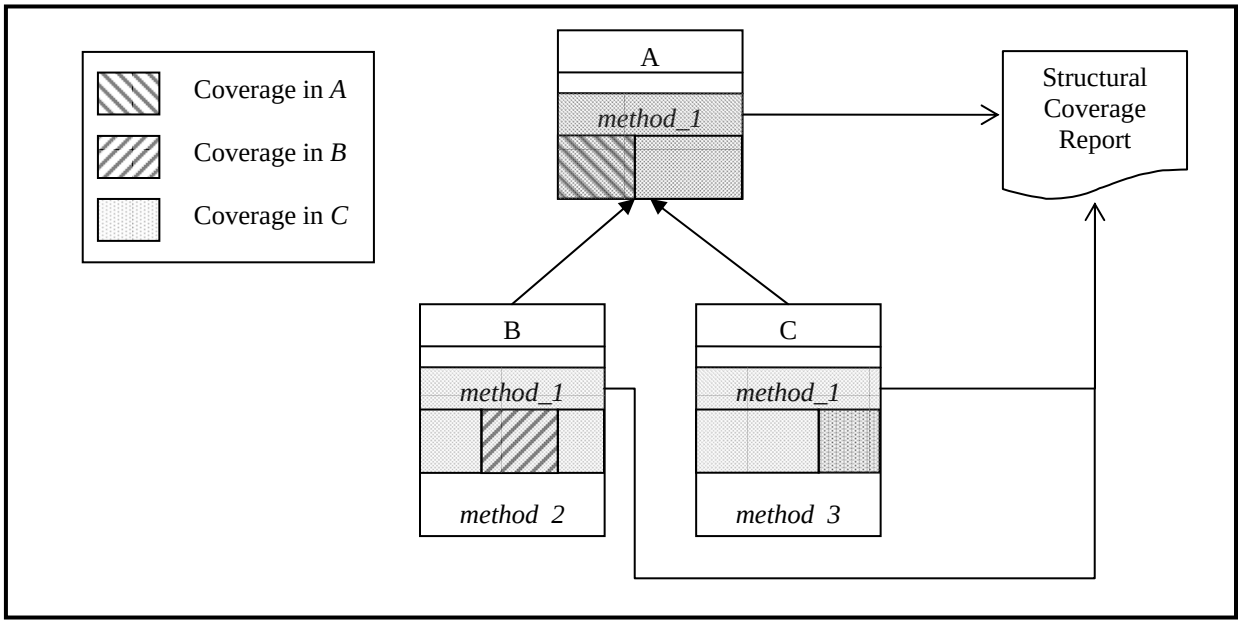
Consider the class diagram in Example A where *method_1()* is inherited by classes *B* and *C*. Suppose portions of *method_1()* (indicated by the shaded regions) are executed in each of *A*, *B*, and *C* such that 100% of *method_1()* is covered, but not in any one context. A report generated at the method level would produce a misleading view of coverage.

Alternately, consider the *flattened* class diagram in Example B. Each method is considered in the context of the flattened class in which it exists. Hence coverage of *method_1()* is reported independently for each context (*A*, *B*, and *C*).

Example A: Structural coverage of concrete (non-inherited) attributes produces a single report purporting 100% coverage of *method_1()*. However, method_1() was not completely covered in any one context.



Example B: Structural coverage of the *flattened* hierarchy considers *method_1()* to be a member of each derived class. Hence an individual coverage report is generated for *method_1()* in the context of classes *A*, *B*, and *C*.



APPENDIX B (COMPLIANCE)

"LDRA_Compliance" lists the rules in this document that can be automatically checked by LDRA. Rules not checked by LDRA will be verified by manual inspection and results captured on checklists.

Note that if other tools are employed to automatically check rules not checked by LDRA, this appendix should be updated to reflect the source of verification