Coding Standards
for
C++ Language

Document Number 832-0536-005
Revision C
CAGE Code 0EFD0

Rockwell Collins

Contract Number None

	NAME	TITLE	APPROVAL
Prepared By:	Allison M. Jones	Preparer	N/A
Approved By:	Daniel J. Renner	Engineering	On File
Approved By:	Richard D. Dykema	Design Quality Assurance and Control Focal	On File

Revision History

Ver	Rev	Release Date	Originator	Reason(s) for Change
001	-	06/28/96	W. J. Ward	Original Rockwell document release. Based on the Allen-Bradley "ACE Project C++ Coding Standard" document.
002	-	11/16/99	J. W. Grace	Added Section 4.2.1, Special Tags, and updated examples accordingly. Numbered and titled examples. Copyright notices updated in examples. Fixed spacing/indenting in examples. NOTE: Reference CMU/FANS CR #160. Changes to this document are too numerous to include all changes in this summary. In deciding whether to use – 002 for a project, it would be beneficial to read the entire document and not rely on change summaries. PVCS revision 1.13
003	-	04/24/00	S. A. Kacena	Update formatting of document. Update "Approval Signatures" section.
004	-	11/29/02	F. Chevalier	CRL 727 taken into account
005	-	03/11/2010	Mary K Muthuramalingam	DLNK000000004 - To create C++ Coding Standards for Level C Software
005	A	2012-04-03	Jamie Nees	CR FUSN00211018 DLCA: Update Coding Standards - Phase 1 ECO-0383829
005	B	2012-05-18	Daniel Renner	FUSN00234924-Updates for Level C development. FUSN00237921-Update based on customer comments. ECO-0390174
005	C	2013-02-01	Daniel Renner	FUSN00297670-Updates to Naming Conventions and Maximum Line Length FUSN00257950-PROCESS: Coding Standard Updates ECO-0425277

Table of Contents

TABLE OF EXAMPLES

1 Introduction

1.1 Purpose

This document presents a coding standard for the C++ programming language. C++ was designed to support the development of high quality, reliable, portable, and reusable software. This document provides guidelines and limits to be used during software development in order to qualify for Level C and D certification.

In order to obtain insight into how to effectively deal with the most difficult aspects of C++, the examples of code those are provided should be carefully studied. This document assumes that the reader is already familiar with the C++ language as defined in *[2]*.

1.2 Applicability

This document should be followed by anyone involved in the design and implementation of software using the C++ language. It will serve as a reference to the implementers, testers and maintainers of the software. This document will also serve as a guide to be used during code reviews. It will be updated as necessary to reflect changes in process methodology and project experience.

Newly written code for a given project will always be audited against this standard. Reused, legacy or third party software will not be required to comply with any mandatory standards.

1.3 Conventions

1.3.1 External Document References

References to functionality outside the scope of this document will refer to the number enclosed in brackets associated with the specific document in References.

1.3.2 Typeface Conventions

This document contains both standards and guidelines. These are denoted with the **[rulename]** in boldface type and in brackets. Standards and guidelines are listed in Appendix B of this document.
In order to make the code more compact, the examples do not always follow the rules. In these cases, the rule that is broken is indicated.

1.3.3 Terminology

Throughout this document the following terminology applies to the use of the words; shall, should, may, will, can, is, and are.

shall: This is the only word, which defines a compulsory, binding requirement.

should, may, will: These are examples of words that are used in the commentary typically
to further explain a requirement(s) or to define a recommendation.

can, is, are: These are examples of words, which are used in commentary to declare a fact or definition.

1.3.4 General Guidelines

[Standard] Every time a rule in boldface is broken, the code should be clearly documented.

Exception: Machine-generated source is not expected to follow this rule. These rules are for manually generated code. It is expected that changes to machine-generated code should be made in the input to the code generator.

Rules shall be followed except where exceptions are noted. Guidelines reflect a preferred method. It is strongly recommended that this method be followed.

Legacy code (older, inherited code) and third-party code may not be expected to follow this standard. New code added to legacy code is expected to follow this Standard.

Machine generated code from code generators, screen designers, etc., again is not expected to follow this standard. Ideally, machine generated code should not be manually modified unless the code being modified is between protected regions that the code generator recognizes. It should be regenerated using the original tool. Code that is manually inserted in these modules should follow as much of the standard as applies.

The coding standards defined in this document are explained with the applicability of the Level of software certification ranging from C-D as defined in [3]. Level E levies no requirements or restrictions on software.

Unrestricted: No restriction or requirement is applicable.

Mandatory: Use of the language element is imperative where the element is appropriate. The code shall use the language element as described.

Recommended: Use of these language elements are recommended but are not compulsory to attain the level of certification.

Restricted: Additional effort is required to assure that use of the feature results in correct and robust implementation of requirements. Any additional conditions required for restricted features as specified in the listed section shall be satisfied.

Prohibited: The code shall not use the designated **C++** language element.

1.4 References

[1] The Annotated C++ Reference Manual - Bjarne Stroustrup and Margaret Ellis, Addison-Wesley 1990, ISBN 0-201-51459-1

[2] The C++ Programming Language, Second Edition - Bjarne Stroustrup, Addison-Wesley 1991, ISBN 0-201-53992-6

[3] Software Considerations in Airborne Systems and Equipment Certification, RTCA/DO-178B. December 1,1992

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

[5] Bjarne Stroustrup. The C++ Programming Language, 3rd Edition. Addison-Wesley, 2000.

[6] RC-ADM-P-008 Rockwell Collins Operating Procedure Copyrights

2 Naming Conventions

Level C-D: **Recommended**

In this section, it is important to distinguish between identifiers and names. A name is that part of an identifier that shows its meaning. An identifier consists of a prefix, a name, and a suffix (in that order). The prefix and the suffix are optional. A suffix may be provided for "C" interfaces to overloaded functions to disambiguate the function name based on the parameters. Since this is only used in the "C" interface, further discussion is unnecessary.

The purpose of naming conventions is to provide the ability to identify the source (subsystem) of a symbol, and if possible its usage (function, variable, type …) and scope (local, global.)

[IdPrefix] The identifier of every globally visible class, enumeration type, type definition, function, constant, and variable in a subsystem is to begin with a prefix that is unique for that subsystem.
Exception: N/A

[PrefixSeparator] Prefixes should be separated from the name by an underscore ('_'), if it is of the same case as the first character of the name. See Exception to rule Names.
Exception: N/A

Names for the following types of objects are to be prefixed:

- Type names (classes, typedefs, enums, structs, unions, etc.,)
- Global variables and constants,
- Function names (not member function names,)
- Preprocessor macros **(#**define),

Names for the following items are not to be prefixed:

- Local or nested Type names (classes, enums, structs, unions, etc.,)
- Local variables and constants, and
- Member names (functions and data) except for constructors and destructors that have identifiers that shall match the class identifier.

Level C-D: **Mandatory**

[Names] Names shall have the following properties:

- Member variables should be prefixed with "m_" and member pointer variables with "m_p". Following the prefix, the rest of the variable should begin with an uppercase letter. (e.g. m_VariableName or m_pPointerName).
- The names of preprocessor Macros are to be entirely uppercase with the words separated by an underscore '_'.

Exception: No exceptions. (At times, an identifier begins with an abbreviation written in uppercase letters, to emphasize the way in which the name is used. Such an abbreviation is considered to be a prefix). Names with words ending in an uppercase letter will use an underscore to separate the words.

Level C-D: **Recommended**

[IdLength] The maximum length of an identifier is 32 characters.
Exception: N/A

There may be many classes in a large project. Because of this, it is important to be careful that name collisions do not occur. One way of preventing collisions is to have strict rules for assigning names to globally visible objects (such as the use of a prefix). In this way, classes from several different class libraries and subsystems may be used at the same time.

Example 1: Exception using compound names

```
int currentIoStream = 1;
const char* functionTitle = "EUA_Special";
```

Example 2: Definition of a class in the class library Generic Editor (ge)

```
class GE_Pane
{
public:
      // Default constructor
      GE_Pane();
      …
private:
      int id;
      …
};
```

Level C-D: **Prohibited**

[Underscore] Developers shall not use identifiers which begin with one or two underscores ('_' or '__').
Exception: Entry points required by the compiler. Enumerations, identifiers, or entry points provided by or required by a third-party library. Note that "third-party library" includes the use of a library produced and provided by another group in Rockwell Collins.

The use of two underscores ('__') in identifiers is reserved for the compiler's internal use according to the ANSI-C standard.
Underscores ('_') are often used in names of library functions (such as "_main" and "_exit"). In order to avoid collisions, do not begin an identifier with an underscore.

[NameAbbrev] Names shall not include ambiguous abbreviations.
Exception: No exceptions.

One rule of thumb is that a name that cannot be pronounced is a bad name. A long name is normally better than a short, cryptic name, but the truncation problem shall be taken into consideration. Abbreviations can always be misunderstood. Global variables, functions and constants ought to have long enough names to avoid name conflicts.

Level C-D: **Recommended**

[NameUsage] Choose variable names that suggest the usage.
Exception: N/A

Classes (objects) should be named so that that "class::function" or "object.function" is easy to read and appears to be logical.

Example 3: Choice of names

```
int groupID; instead of grpID
int nameLength; instead of namLn
PrinterStatus resetPrinter; instead of rstprt
```

Example 4: Ambiguous names

```
void termProcess(); terminate Process or terminal Process?
```

Example 5: Names having numeric characters can cause errors, which are difficult to locate.

```
int I0 = 13; Names with digits can be difficult to read.
int IO = I0; {letter 'O' or a zero?}
```

[Globals] Encapsulate global functions, variables, constants, enumerated types, and typedefs in a class.
Exception: Global variables may exist in the name space if they have a subsystem prefix.

The use of prefixes can sometimes be avoided by using a class to limit the scope of the name. Static variables in a class should be used instead of global variables and constants, enumerated data types, and typedefs. Although nested classes may be used in C++, these give rise to too many questions (in connection with the language definition) to be able to recommend their use.

Example 6: One way to avoid global functions and classes

Instead of declaring :

```
void GE_MyFunc1();
void GE_MyFunc2();
```

Encapsulate the functions using an abstract class:

```
class GE_GenericEditor
{
public:
      static void MyFunc1();
      static void MyFunc2();
private:
      virtual Dummy() = 0; Technique used to make the class abstract
};
```

Now, they may be accessed by using the scope-operator:

```
GE_GenericEditor::MyFunc1();
GE_GenericEditor::MyFunc2();
```

3 Source Code in Files

3.1 Naming Files

Level C-D: **Mandatory**

The purpose of these conventions is to provide a uniform interpretation of file names.

[IncludeExt] Include files in C++ shall always have the file name extension ".h".
Exception: No exceptions.

[SourceExt] Implementation files in C++ shall always have the file name extension ".cpp".
Exception: No exceptions.

[ClassDef] An include file shall not contain more than one class definition.
Exception: Closely coupled classes such as helper classes, which are not referenced by external classes/functions, may coexist with the user class.

Level C-D: Recommended

[ClassDefFileName] An include file for a class should have a file name of the form <class identifier>.h.
Exception: N/A

The class definition file contains the class interface; definition, and its inline functions.

[ClassImpl] A class implementation should be in one file.
Exception: N/A

When tools for managing C++ code are not available, it is much easier for those who use and maintain classes if there is only one class definition in each file and if implementations of member functions in different classes are not present in the same file.
It is especially important to remember that virtual functions are always linked. Since most compilers refer to these via virtual tables or "vtables".

[ClassImpl2] A class implementation that is very large (greater than 3000 lines), should be reviewed for opportunities to introduce inheritance.
Exception: N/A

It is possible that some of the functionality of the class may be moved into another to not only make the class size more manageable, but also to allow for reuse of common functionality.

[FileNames] Always give a file a name that is unique in as large a context as possible.
Exception: N/A

There is always a risk for name collisions when the file name is part of identifier names that are generated by the compiler. This may be a problem in using some compilers. It is easily understood that if a program has two files with the same name but in different sub directories, there may be name collisions between the functions generated. Since class names shall generally be unique within a large context, it is appropriate to utilize this characteristic when naming its include file. This convention makes it easy to locate a class definition using a file-based tool.

3.2 Comments and Headers

3.2.1 Comments

Level C-D: **Mandatory**

[Comments] All comments shall be written in English.
Exception: No exceptions.

It is necessary to document source code. This should be compact and easy to find. By properly choosing names for variables, functions, and classes and by properly structuring the code, there is less need for comments within the code. Note that comments in include files are meant for the users of classes, while comments in implementation files are meant for those who maintain the classes.

Comments are 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, and is placed, if possible, at the end of that line. Unfortunately, too many tactical comments can make code unreadable. For this reason, it is recommended to primarily use strategic comments, unless trying to explain very complicated code.

Comments in the implementation files should be placed within the scope of the method that is being documented.

Comments in header files should be placed in the line(s) immediately preceding the line of code being documented (class, method, or attribute). This will make it possible for the design modelling tool to do two things, automatically retrieve comments from the code and place these comments into a model, and place comments from a model into the source files during code generation.

The preferred method to remove code for compilation is "/* ... */" blocks. With the current code coverage tools, "#if 0 … #endif" blocks are treated as active or compiled in code, and then result in "uncovered code" that must be explained away before final release of the Structural Coverage Analysis. If the characters "//" are consistently used for writing comments, then the combination "**/* … */**" may be used to make comments out of entire sections of code during the development and debugging phases.

Commentary: It is assumed the number of slashes creating the border is consistent to the size of the page, consistent throughout the code and not meant to be a hard number, such as "80 slashes are required".

Example 7: Strategic and Tactical Comments

```
int insanelySmallAndSimpleFunction(int i)
{
      // This is a strategic comment describing the following block of code
      {
      …
      }
…
}
int insanelyGreatAndComplicatedFunction(int i)
{
      int index = i++ + ++i * i-- - --i; // This is a tactical comment
      return index;
}
```

Example 8: Comments in Header Files

```
// this comment will come from (or will be inserted into) the
// documentation field of the String class item in the design model tool
```

```
class String
{
private:
      // this comment will come from (or be inserted into) the
      // documentation field of the attribute stringLen
      int stringLen;
public:
      // this comment will come from (or be inserted into) the
      // documentation field of the method getText
      char* getText();
      // this comment will come from (or be inserted into) the
      // documentation field of the method getLength
      int getLength()
      {
            // this comment will not appear in the model
            return stringLen;
      };
};
```

3.2.2 Copyright Notice

Level C-D: **Mandatory**

[Copyright] All code shall be copyrighted. If the code has been developed over a period of years, each year shall be stated. All code from third-party sources that include copyright notices shall have the Rockwell Collins copyright notice added preceding it in the file header. The date should encompass date of first and last modification and all years in between in which changes were released through the formal release process.

See the Rockwell Collins Operating Procedure Copyrights [6] for current copyright policy. This policy is the official company policy on how to document copyrights.

Exception: No exceptions.

Example 9: Copyright Notice

```
////////////////////////////////////////////////////////////////////////////
//
//(c) Rockwell Collins, 20xx, 20xx-20xx All Rights Reserved.
//
////////////////////////////////////////////////////////////////////////////
```

3.3 Include Files

Level C-D: **Mandatory**

[NestIncludes] Every include file shall contain a mechanism that prevents multiple inclusions of the file.
Exception: No exceptions.

The easiest way to avoid multiple includes of files is by using an “#ifndef/#define” block in the beginning of the file and an “#endif” at the end of the file.

Example 10: Technique for preventing multiple inclusion of an include file

```
#ifndef FOO_H
#define FOO_H
      // The rest of the file
#endif
```

[IncludeH] When the following kinds of definitions are used (in implementation files or in other include files), they shall be included as separate include files:

- classes that are used as base classes,
- classes that are used as member variables,
- classes that appear as return types or as argument types in function/member function prototypes.
- function prototypes for functions/member functions used in inline member functions that are defined in the file.

Exception: No exceptions.

Level C-D: Recommended

[IncludeCpp] Every implementation file is to include the relevant files that contain:

- declarations of types and functions used in the functions that are implemented in the file.
- declarations of variables and member functions used in the functions that are implemented in the file.

Exception: N/A

The number of files included by a file should be minimized. If a file is included in an include file, then every implementation file that includes the second include file shall be re-compiled whenever the first file is modified. A simple modification in one include file can make it necessary to re-compile a large number of files.

Level C-D: Mandatory

[RefByName] Definitions of classes that are only accessed via pointers (*) or references (&) shall not be included as include files.

Exception: No exceptions.

When only referring to pointers or references to types defined in a file, it is often not necessary to include that file. It may suffice to use a forward declaration to inform the compiler that the class exists. Another alternative is to precede each declaration of a pointer to the class with the keyword “class”. If a file only contains information that is only needed in an implementation file, that file should not be included in another include file. Otherwise, when the information is no longer needed in the implementation file, it may be necessary to re-compile each file that uses the interface defined in the include file.

Example 11: Include file for the class PackableString

file: PackableString.h

```
#ifndef PACKABLESTRING_H
#define PACKABLESTRING_H

#include “String.h”
#include “Packable.h”
```

It is not necessary to extern-declare class Buffer when each pointer declaration specifies the keyword class as shown below.

An explicit extern-declaration makes the code easier to understand.

```
class Buffer;
class PackableString : public String, public Packable
{
public:
      PackableString(const String& s);
      class Buffer* Put(Buffer* outbuffer);
      …
};
```

```
#endif
```

Example 12: Implementation file for the class PackableString

PackableString.cpp

```
#include "PackableString.h"
```

To be able to use Buffer-instances, Buffer.h is included.

```
#include "Buffer.h"
Buffer*
PackableString::Put(Buffer* outbuffer)
{
…
}
```

Level C-D: Recommended

[Paths] Do not specify relative paths in #include directives.
Exception: When using an IDE with a well-defined file organization.

True portable code is independent of the underlying operating system. For this reason, paths should be avoided when including files. The processing of such search paths depends on the compiler and system environment should not be taken for granted. Instead, search paths should be provided in 'mak**e'** files as options for the compiler.

Example 13: Never use explicit path names

Wrong:

```
#include "..\include\fnutt.h"
```

Wrong:

```
#include <sys\socket.h>
```

Level C-D: **Mandatory**

[IncludeForm] The directive #include "filename.*h"* shall be used for ALL include files.
Exception: Standard header files that are provided by the compiler such as iostream.h, etc. should be included using #include <filename.h>.

3.4 Source Code Presentation

Level C-D: **Recommended**

[Presentation] Indentation should be three spaces for each level. Spaces should be used; tabs should not be included in source code. Blank lines or stategic comments will be included to separate major blocks of code from each other.
Exception: N/A

4 Language Considerations

4.1 Style

4.1.1 Formatting

4.1.1.1 Functions

Level C-D: **Recommended**

[FuncDef] When defining functions, the complete function definition will be on the same line including return type, function name, and any arguments.
Exception: If there is not enough room, continuing arguments will be entered on the next line spaced over so that they start underneath the previous line's arguments. If it is desired to comment each argument then each argument shall appear on a separate line with a comment following it.

Example 14: Right way for declaring a function (in function definition)

Correct:

```
int myComplicatedFunction(unsigned unsignedValue,          // first comment
                          int intValue,                    // second comment
                          char* charPointerValue,          // third comment
                          int* intPointerValue,            // fourth comment
                          myClass* myClassPointerValue,    // fifth comment
                          unsigned* unsignedPointerValue); // sixth comment
```

4.1.1.2 Compound Statements

Level C-D: **Recommended**

[Indent] Blocks of code will be indented three spaces.
Exception: N/A

[Braces] Braces ("{}") which enclose a block are to be placed in the same column, indented to the depth of the preceding code, on separate lines directly before and after the block.
Exception: N/A

The placement of braces seems to have been the subject of the greatest debate concerning the appearance of both C and C++ code. The style that has been selected gives the most readable code. Other styles may well provide more compact code. The braces will be indented to the same depth as the enclosing code.

Example 15: Brace position and Block Indentation

Indented Braces:

```
if (a == b)
{
   err("b");
   ++nerrs;
}
```

4.1.1.3 Data Formatting

Level C-D: **Recommended**

[DataFormat] Data should be formatted to reflect the logical organization.
Exception: N/A

Data should look like its structure. Initialization of a 2-dimensional matrix should look like a rectangle. A vector would be a single line. Data that is logically coupled like coordinates can be declared on one line. In addition the braces should in general match that of the code.

4.1.1.4 Flow Control Statements

Level C-D: **Recommended**

[EmptyFlow] The flow control primitives if, else, while, for and do should be followed by a block, even if it is an empty block.
Exception: N/A

[SimpleFlow] The expression controlled by the flow control primitives if, else, while, for and do should be placed on a new line.
Exception: N/A

At times, everything that is to be done in a loop may be easily written on one line in the loop statement itself. It may then be tempting to conclude the statement with a semicolon at the end of the line. This may lead to misunderstanding since, when reading the code, it is easy to miss such a semicolon. It seems to be better, in such cases, to place an empty block after the statement to make completely clear what the code is doing.

Example 16: Flow control structure without statements

No block at all - No

```
while ( /* Something */ );
```

Empty block - better

```
while ( /* Something */ )
;
```

Empty block - best

```
while ( /* Something */ )
{    // Empty
}
```

4.1.1.5 Operators

Level C-D: **Recommended**

[AddressTypes] The dereference operator **'*'** and the address-of operator '&' should be directly connected with the type names in declarations and definitions.
Exception: N/A

The characters '*' and '&' should be written together with the types of variables (int* i) {instead of with the names of variables (int *i)} in order to emphasize that they are part of the type definition. This is the preferred coding style. Instead of saying that *i is an int, say that i is an int*.

Variable declarations will be limited to one per statement as stated in Rule VarDecl in section Variables.

Traditionally, C recommendations indicate that '*' should be written together with the variable name, since this reduces the probability of making a mistake when declaring several variables in the same declaration statement (the operator '*' only applies to the variable on which it operates). Since the declaration of several variables in the same statement is not recommended (see VarDecl), however, such advice is not needed.

Example 17: Operators * and & together with the type

```
char* Object::AsString()
{
      // Something
};
char* userName = 0;
int sfBook = 42;
int& anIntRef = sfBook;
```

Example 18: Declaration of several variables in the same statement

Avoid declarations like this: it is not a pointer.

```
char* i,j; // i is declared pointer to char, while j is declared char
```

4.1.2 Bit Field Declarations

Level C-D: **Mandatory**

[BitField] Bit Fields shall have explicitly unsigned integral or enumeration types only. The bit field definition shall be platform independent. It supports both big endianess and little endianess.
Exception: No exceptions.

Explicitly declaring a bit-field unsigned prevents unexpected sign extension or overflow. Bit field declarations should specify size.

4.1.3 Data Definitions

Level C-D: **Mandatory**

[DataDef] Data definitions shall be explicit, i.e. long, short, char. Do not use int or unsigned without an explicit qualifying long, short, char, etc.
Exception: No exceptions.

Level C-D: **Mandatory**

[Union] Any data type defined as a "union" shall be platform independent. Byte ordering shall be transparent to both endianess.
Exception: No exceptions.

Example 19: Byte ordering in Union with Endianess

```
typedef struct A429DATA
{
#if defined (SMALLEND)
  char   byte2;
  char   byte3;
  char   byte4;
```

```
#else  //Big Endian
  char   byte4;
  char   byte3;
  char   byte2;
#endif /*(endian-ness) */
} A429DATA;


typedef union A429WORD
{
  unsigned int full;

  struct
  {
#if defined (SMALLEND)
    char      label;
    A429DATA  data;
#else
    A429DATA  data;
    char      label;
#endif /* (endian-ness) */
  } raw;

  struct
  {
#if defined (SMALLEND)
    char label;
    char byte2;
    char byte3;
    char byte4;
#else
    char byte4;
    char byte3;
    char byte2;
    char label;
#endif /*  (endian-ness) */
  } data;
}
```

4.1.4 Classes

Level C-D: **Mandatory**

[AccessControl] The public, protected, and private sections of a class shall be declared in that order (the public section is declared before the protected section, which is declared before the private section).

Exception: No exceptions.

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 have the least general interest. The access qualifier tags only need to be present if there is a class member of that access type

Example 20: A class definition in accordance with the style rules

```
class String : private Object
{
public:
      // Default constructor
```

```
        String();
        // Copy constructor
        String(const String& s);
        unsigned Length() const;
        …
protected:
        int CheckIndex(unsigned index) const;
        …
private:
        unsigned noOfChars;
        …
};
```

4.1.5 Functions

Level C-D: **Recommended**

[Return] Always specify the return type of a function explicitly.
Exception: Constructors and destructors do not have any return type associated with them.

Functions, for which no return type is explicitly declared, implicitly receive int as the return type. This can be confusing for a beginner, since the compiler gives a warning for a missing return type. Because of this, functions that return no value should specify void as the return type. Also this defies the concept that all information is explicitly present in the definition and implementation.

Example 21: Functions that return no value should have the return type void.

```
void StrangeFunction(const char* before, const char* after)
{
        …
}
```

4.2 Classes

4.2.1 Data Encapsulation

Level C: **Mandatory**
Level D: **Recommended**

[NoPubAttr] A class shall not have public attributes.
Exception: Static constant data members are exempt from this rule.

Object oriented principles dictate that exposing data members is not a desired design choice. In general, in order to respect the data encapsulation principles, all the attributes of a class shall be private. However, in embedded systems, certain concessions are made. Since static constant data members are set once and not changed, making those attributes public is acceptable.

4.2.2 Standard Class Template

Level C-D: **Mandatory**

[StdClass] All classes shall define a default constructor, a copy constructor, an assignment operator (operator=()), and a destructor.
Exception: No exceptions

If these functions are not declared, the compiler will generate these functions. The problem is that the compiler will not necessarily do what is really wanted. This is especially true if pointers to free store data are kept in the class. The compiler-generated copy constructor and assignment operator do bitwise copies, which could then lead to later memory problems. This rule follows the general guideline that all functionality in the class should be explicitly stated. The default constructor traditionally is a constructor with no arguments. It is now permissible to have a default constructor that takes arguments as long as all the arguments have default values. There can only be one default constructor in a class. Functions that are not intended to be used must be declared in the private section of the class declaration, and DO NOT implement the function. This will safeguard against its use by the compiler issuing a compile error for any class without access to private functions. Additionally, for classes that have access to the private functions, the linker will issue a linker error that the particular function is not implemented.

Example 22: Standard Class Template

```
class Message
{
public:
the public user interface to a Message:
      …
      // basic constructor – a Message requires a string to be constructed
      Message (char*);
      // copy constructor
      Message (const Message&);
      // assignment operator
      Message& operator=(const Message&);
      // destructor
      ~Message ();
private:
      // default constructor-don't use-not implemented
      Message ();

};
```

4.2.3 Constructors and Destructors

Level C-D: **Recommended**

[InitVars] Member variables should be defined and initialized in the same order.
Exception: N/A

Member variables are always initialized in the order they are declared in the class definition; the order in which you write them in the constructor initialization list is ignored.

Example 23: Initializing member variables in order they are declared

```
class employee
{
private:
string email, first_name, last_name;
public:
employee(const char* first, const char* last) :
first_name(first),last_name(last),email(first_name+last_name+"@collins.com"
){}
};
```

Because the member 'email' is declared first, it will be initialized first and will attempt to use the other not-yet initialized members first. The solution is to always initialize member variables in the same order in which they were declared.

Many compilers will issue a warning if you break this rule.

Level C-D: **Mandatory**

[InitList] Initialization shall be preferred to assignment in constructors
Exception: No exceptions.

Using initialization instead of assignment to set member variables takes about the same amount of typing and all other things being equal, the code will run faster.

Level C-D: **Mandatory**

[VirtDestr] Base class destructors shall be made public and virtual, or protected and non-virtual
Exception: No exceptions

If deletion through a pointer to a base class should be allowed, then the base class's destructor must be public and virtual. Otherwise, it should be protected and non-virtual.

Level C-D: **Prohibited**

[ExtLink] Entities shall not be defined with linkage in a header file.
Exception: No exceptions

Repetition causes bloat: Entities with linkage, including namespace-level variable or functions, have memory allocated for them. Defining such entities in header files results in link-time errors or memory waste. Put all entities with linkage in implementation files.

Example 24: Entities must not be defined with linkage in header files

```
// avoid defining entities with external linkage in a header
int fudgeFactor;
string hello("Hello, world!");
void foo() {/* … */}
```

This is liable to cause link-time errors complaining of duplicate symbols as soon as such a header is included by more than one source file. The reason is simple: Each source file actually defines and allocates space for fudgeFactor and hello and foo's body, and when the time comes to put it all together (linking), the linker is faced with several symbols bearing the same name and competing for visibility.

The solution is simply put just the declarations in the header:

```
extern int fudgeFactor;
extern string hello;
void foo();                 // "extern" is optional with function declarations
```

Level C-D: **Prohibited**

[GlobalsInCtor] Global objects shall not be used in constructors and destructors.
Exception: No exceptions.

Level C-D: **Prohibited**

[StaticInit] It shall not be assumed that static objects are initialized in any special order.
Exception: No exceptions.

In connection with the initialization of statically allocated objects, it is not certain that other static objects will be initialized (for example, global objects, or a static object that was declared external). This is because the order of initialization of static objects, which is defined in various compilation units, is not defined in the language definition.

There are ways of avoiding this problem, all of which require some extra work.

Example 25: Dangerous use of static objects in constructors

Hen.h
```
class Egg;
class Hen
{
public:
      // Default constructor
      Hen();
      // Destructor
      ~Hen();
      …
      void MakeNewHen(Egg*);
      …
};
```
Egg.h
```
class Egg { };
extern Egg theFirstEgg; // defined in Egg.cpp
```
FirstHen.h
```
class FirstHen : public Hen
{
public:
      // Default constructor
      FirstHen();
      …
};
extern FirstHen theFirstHen; // defined in FirstHen.cpp
```
FirstHen.cpp
```
FirstHen theFirstHen; // FirstHen::FirstHen() called
FirstHen::FirstHen()
{
The constructor is risky because theFirstEgg is a global object and may not yet
exist when theFirstHen is initialized. Which comes first, the chicken or the egg
?
      makeNewHen(&theFirstEgg);
}
```

Level C-D: **Prohibited**

[VirtualInCtor] Calling virtual functions in a base constructor shall be prohibited.
Exception: No exceptions.

If virtual functions in a derived class are overridden, the original definition in the base class will still be invoked by the base class' constructor. Override, then, does not always work when invoking virtual functions in constructors.

Example 26: Override of virtual functions does not work in base class constructors

```
class Base
{
public:
      // Default constructor
      Base();
      virtual void Foo() { cout << "Base::foo" << endl; }
      …
};
```

```
Base::Base()
{
      Foo(); // Base::foo() is ALWAYS called.
}
// Derived class overrides foo()
class Derived : public Base
{
public:
      virtual void Foo() { cout << "Derived::foo" << endl;} //foo is
      // overridden
      …
};
main()
{
      Derived d;
      // Base::Foo() called when the Base-part of
      // Derived is constructed.
}
```

Level C-D: Recommended

[CtorInitOrder] Do not assume that an object is initialized in any special order in constructor initialization list.
Exception: N/A

The order of initialization for items in a constructor initialization list is based on the order of the objects in the class definition, not on the order of the items in the initializer list. This can lead to interesting problems.

The order of initialization for static objects may present problems. A static object may not be used in a constructor, if it is not initialized until after the constructor is run. At present, the order of initialization for static objects, which are defined in different compilation units, is not defined. This can lead to errors that are difficult to locate. There are special techniques for avoiding this.

Example 27: Do not depend on the order of initialization in constructors.

```
#include <iostream.h>
class X
{
public:
      X(int y);
private:
      int i;
      int j;
};
inline X::X(int y) : j(y), i(j) No. j may not be initializedbefore i !!
{
      cout << "i:" << i << " & " << "j:" << j << endl;
}
main()
{
      X x(7); Rather unexpected output: i:0 & j:7
}
```

The destructor is a member function that in most cases should be declared virtual. It is necessary to declare it virtual in a base class if derived class objects are deleted through a base class pointer. If the destructor is not declared virtual, only the base class destructor will be called when an object is deleted that way.

4.2.4 Assignment Operators

Level C-D: **Mandatory**

[OpAssign] An assignment operator that performs a destructive action shall be protected from performing this action on the object upon which it is operating.
Exception: No exceptions.

An assignment is not inherited like other operators. If an assignment operator is not explicitly defined, then one is automatically defined instead. Such an assignment operator does not perform bit-wise copying of member data; instead, the assignment operator (if defined) for each specific type of member data is invoked. Bit-wise copying is only performed for member data having primitive types.

One consequence of this is that bit-wise copying is performed for member data having pointer types. If an object manages the allocation of the instance of an object pointed to by a pointer member, this will probably lead to problems: either by invoking the destructor for the managed object more than once or by attempting to use the deallocated object. If an assignment operator is overloaded, the programmer shall make certain that the base classes and the members' assignment operators are run.

A common error is assigning an object to itself (a = a). Normally, destructors for instances that are allocated on the heap are invoked before assignment takes place. If an object is assigned to itself, the values of the instance variables will be lost before they are assigned. This may well lead to strange run-time errors. If a = a is detected, the assigned object should not be changed.

Level C-D: **Recommended**

[AssignReturn] An assignment operator ought to return a const reference to the assigning object.
Exception: N/A

If an assignment operator returns void, then it is not possible to write a = b = c. It may then be tempting to program the assignment operator so that it returns a reference to the assigning object.
Unfortunately, this kind of design can be difficult to understand. The statement (a = b) = c can mean that a or b is assigned the value of c before or after a is assigned the value of b. This type of code can be avoided by having the assignment operator return a const reference to the assigned object or to the assigning object. Since the returned object cannot be placed on the left side of an assignment, it makes no difference which of them is returned (that is, (a = b) = c is no longer correct).

Example 28: Incorrect and correct return values from an assignment operator

```
void MySpecialClass::operator=(const MySpecialClass& msp);
```
Well ...?
```
MySpecialClass& MySpecialClass::operator=(const MySpecialClass& msp);
```
No
```
const MySpecialClass& MySpecialClass::operator=(const MySpecialClass& msp);
```
Recommended

Example 29: Definition of a class with an overloaded assignment operator

```
class DangerousBlob
{
public:
      const DangerousBlob& operator=(const DangerousBlob& dbr);
      …
private:
      char* cp;
};
```

```
//Definition of assignment operator
const DangerousBlob&
      DangerousBlob::operator=(const DangerousBlob& dbr)
{
      if (this != &dbr) Guard against assigning to the this pointer
      {
            delete [] cp; Disastrous if this == &dbr
      }
      …
}
```

4.2.5 Member Function Return Types

Level C-D: **Restricted**

[PublicReturn] A public member function should not return a pointer or non-const reference to member data.
Exception: N/A

[PublicReturnExt] A public member function should not return a pointer or non-const reference to data outside an object, unless the object shares the data with other objects.
Exception: N/A

General Object-Oriented design principles state that by allowing a user direct access to the private member data of an object, this data may be changed in ways not intended by the class designer. This may lead to reduced confidence in the designer's code: a situation to be avoided. However, in the development of embedded systems, there are situations where this principle can be compromised in the interest of performance. This design decision should be reviewed with experienced engineers. In reality, this is no different than a C program returning a pointer to a calling function. With proper design and review, this is an allowable technique.

This rule also aims to avoid a calling function having a pointer to an object that may become deallocated.

Note that it is not forbidden to use protected member functions that return a const reference or pointer to member data.

Example 30: Returning a non-const reference to member data from a public function.

```
class Account
{
public:
      Account(int myMoney) : moneyAmount(myMoney) {};
      const int& GetSafeMoney() const { return moneyAmount; }
      int& GetRiskyMoney() const { return moneyAmount; } // returns access
      …
private:
      int moneyAmount;
};
Account myAcc(10);
myAcc.GetSafeMoney() += 1000000; // Compilation error: assignment to constant
myAcc.GetRiskyMoney() += 1000000; // myAcc::moneyAmount = 1000010
```

4.2.6 Member Functions in derived Classes

Level C: **Mandatory**
Level D: **Unrestricted**
Exception: No exceptions.

[Polymorphism] All public and protected functions in a base class or an intermediate class shall be declared as virtual or pure virtual.
Note: Assignment Operators need not be virtual or pure virtual.

Once a virtual public or virtual protected function (this includes pure virtual) has been implemented, all classes derived from that class must also implement the function. Note that if the design changes such that a class transitions to a base class, all of its functions must be converted to virtual or pure virtual.

For a derived class where the base class function implementation is fully sufficient, the derived class' override function will only explicitly call the base class function. If a class is an intermediate class in the hierarchy, and one of its base classes contains a pure virtual function, it is not required for the developer to implement the pure virtual function in the intermediate class to satisfy the polymorphism concern. The intermediate class is still considered a base class with a pure virtual function for the purposes of this standard.

This strategy allows structural coverage tools to clearly demonstrate the use of inherited functions by all derived classes to allow structural coverage tools to demonstrate Control Coupling.

The derived class' re-direct to the base class' function should be implemented as an in-line code statement following the declaration of the function in the derived class' header file. This sets this type of code as unique to indicate that it is only a re-direct to the base class function. See examples below:

The reason for this standard is because for DO-178B Level C and higher projects this satisfies DO-178B Table A-7, objective 8, without compliance to this a manual analysis is required or a tool that can prove control coupling.

Example 31: Base, Intermediate and Derived classes

IMsgAgent.h : Base class Protected Function operator= has public/protected access but is not virtual.

```
//Hide forbidden assignment operator
   const IMsgAgent& operator=(const IMsgAgent&);
```

The above case is default assignment operator present in protected section. This cannot be virtualized.

Example 32: Base, Intermediate and Derived classes

[**Base Class**] eat() and breathe() are pure virtual functions

```
class Animal {
public:
      virtual void eat() = 0;             //Pure Virtual Function
      virtual void breathe() = 0;   //Pure Virtual Function
};
```

[**Intermediate class**] Base class for Minnow and Walleye

```
class Fish: public Animal {
public:
      //Virtual function and Implementation of base class function
      virtual void eat() {std::cout << "Fish eating plants...\n";};

      //Virtual function and Implementation of base class function
      virtual void breathe(){std::cout << "Fish breathing in water...\n";}
};
```

[**Derived Class**]

```
class Minnow: public Fish {
```

```
public:
      void eat(){Fish::eat();};      // explicit call to the Intermediate base
class function.
      void breathe(){Fish::breathe();};     // explicit call to the Intermediate
base class function.
};

class Walleye: public Fish {
public:
      //Override of Intermediate base class function
      void eat() {std::cout << "Walleye eating Minnows...\n";};
      void breathe(){Fish::breathe();};     // explicit call to the Intermediate
base class function.
};
```

4.2.7 Inheritance

Level C-D: **Restricted**

[InheritMult] Multiple inheritance will be considered a restricted design method**.** Multiple Inheritance should be used carefully and after proper review by experienced engineers.
Exception: N/A

Multiple inheritance should be avoided. Its use should be limited so that inheritance paths remain reasonably simple. The inheritance hierarchy should remain reasonably shallow.
The main concern with multiple inheritance where deep inheritance hierarchies are present is the risk of introducing the "deadly diamond" problem. This is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If D calls a method defined in A (and does not override the method), and B and C have overridden that method differently, then it is unclear which method is called.
With adherence to [Polymorphism] rule, then this risk is mitigated, as the classes B and C described above must declare their functions properly, and the derived class D must then explicitly call out the B or C function called.

Level A-D: **Mandatory**

[VirtualDtor] All classes shall have a virtual destructor.
Exception: If the base class does not have a virtual destructor then all subclasses of that class cannot have destructors.

Classes that will be used as a base class will still follow the standard class template with one exception. This exception is that the class destructor will be made a virtual function. This will allow child classes destroyed properly through pointers to the base class.

Example 33: Standard Base Class Template

```
class Base
{
public:
      the public user interface to a Base:
      …
      // default constructor
      Base();
      // copy constructor
      Base(const Base&);
      // assignment operator
      Base& operator=(const Base&);
      // virtual destructor
```

```
      virtual ~Base();
      #ifdef TEST || DEBUG
      check_class();
      #endif
};
```

4.2.8 Templates

Templates provide a powerful technique for creating families of functions or classes parameterized by *type*. Templates provide for reusability in programming languages. As a result, generic components may be created that match corresponding hand-written versions in both size and performance.
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 *[5]*, 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.

Level C: **Mandatory**
Level D: **Unrestricted**

[TmpCust] Templates shall be customized intentionally and explicitly.
Exception: No exceptions.

When writing a template, provide points of customization knowingly and correctly, and document them clearly. When using a template, know how the template intends, and customize it appropriately.
There are three major ways to provide points of customization in a template

The first way to provide a point of customization is the usual "implicit interface" approach where the template simply relies on a type's having an appropriate member with a given name:

Example 34: Using "implicit interface"

```
template<typename T>
void Sample1( T t ) {
t.foo();                    // foo is a point of customization
typename T::value_type x; // another example: providing a point of
}                           //customization to look up a type (usually via
                            //typedef)
```

The second option is to use the "implicit interface" method, but with a nonmember function that is looked up via argument-dependent lookup (i.e., it is expected to be in the namespace of the type with which the template is instantiated).

Example 35: Using implicit interfaceWith Non member function

```
template<typename T>
void Sample2( T t ) {
  foo( t );             // foo is a point of customization
  cout << t;             // another example: operator<< with operator the same
}                       // notation is kind of point of customization
```

The third option is to use specialization, so that the template is relying on a type's having specialized (if necessary) another class template provided.

Example 36: Using Specialization for customization

```
template<typename T>
void Sample3( T t ) {
  typename S3Traits<T>::foo( t );     // S3Traits<>::foo is a point of
customization

  typename S3Traits<T>::value_type x; // another example: providing a point of
  }                                   // customization to look up a type
                                      //(usually via typedef)
```

Level C: **Recommended**
Level D: **Unrestricted**

[FuncTmp] Do not specialize function templates.

Exception: N/A

Specialization is good only when it can be done correctly. When extending someone else's function template, avoid trying to write a specialization, instead, write an overload of the function template and put it in the namespace of the type(s) the overload is designed to be used for. When writing your own function template, avoid encouraging direct specialization of the function template itself.
Function templates cannot be specialized partially, only totally. Function template specializations never participate in overloading, so any specialization that is written will not affect the template which gets used.

Example 37: Do not specialize function templates.

```
template<class T> // (a) base Template
void f( T );

template<>        // (c) explicit specialization, of (a)
void f<>(int*);

template<class T> // (b) a second base template, overloads (a)
void f( T* );

// ...

int *p;
f( p );           // calls (b) overload resolution ignores
                  // specializations and operates on the base
                  // function templates only
```

The key to understanding this is simple, and it is- Specializations don't overload. So do not specialize function templates.

Level C: **Mandatory**
Level D: **Unrestricted**

[TmpSpl] A template specialization shall be declared before its use.
Exception: No exceptions.

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

Example 38: Template Specialization declaration

```
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.
```

Level C: **Recommended**
Level D: **Unrestricted**

[TmpDependence] A template definition's dependence on its instantiation contexts should be minimized.
Exception: N/A

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

Level C: **Recommended**
Level D: **Unrestricted**

[SplPtr] Specializations for pointer types should be made where appropriate.
Exception: N/A

Pointer types often require special semantics or provide special optimization opportunities

Level C: **Recommended**
Level D: **Unrestricted**

[MulDef] Avoid multiple definitions of overloaded functions in conjunction with the instantiation of a class template.
Exception: N/A

It is not possible in C++ to specify requirements for type arguments for class templates and function templates. This may imply that the type chosen by the user, does not comply with the interface as required by the template. For example, a class template may require that a type argument have a comparison operator defined.

Another problem with type templates can arise for overloaded functions. If a function is overloaded, there may be a conflict if the element type appears explicitly in one of these. After instantiation, there may be two functions which, for example, have the type int as an argument. The compiler may complain about this, but there is a risk that the designer of the class does not notice it. In cases where there is a risk for multiple definitions of member functions, this must be carefully documented.

Example 39: Problem when using parameterized types (Cfront 3.0 or other template compiler)

```
template <class ET>
class Conflict
  {
     public:
        void foo( int a );
        void foo( ET a ); // What if ET is an int or another integral type?
                          // the compiler will discover this
  };
```

Level C: **Prohibited**
Level D: **Unrestricted**

[NestedTemplates] Templates shall not be nested and each instance of a template shall be tested. Each instance of a template with a unique set of arguments should be tested.
Exception: No exception

Note that a template is not compiled, only the instantiation of the template is compiled. The syntax of the template definition is not checked until you it is instantiated. Since a good compiler will avoid compiling parts of a template that aren't actually used, a full check of syntax errors will not be got unless a complete test of the template instantiation is performed.

A template definition cannot be split into a header file and an implementation file with the idea that the implementation file will be separately compiled and only the header file needs to be included where needed. This is a consequence of the paragraph above. All code defining the template must be in the included one file.

Note that the template arguments in a function template must appear as an argument of the function. The return type doesn't count.

When overloaded operators are defined, remember that they keep their associativity and precedence. Sometimes this means that a desirable operator that is based on the symbols that define it would be inappropriate and confusing in its usage. In particular note that operator "<<" and operator ">>" have middle level precedence (between the arithmetic and the comparison operators). The assignment operators have a lower precedence than the "<<" and “>>" operator, which is sometimes confusing when you are doing input and output. Also, the assignment operators are right associative.

4.3 Functions

Unless otherwise stated, the following rules also apply to member functions.

A function must never return, or in any other way give access to, references or pointers to local variables outside the scope in which they are declared. Returning a pointer or reference to a local variable is always wrong because it gives the user a pointer or reference to an object that no longer exists.

4.3.1 Function Arguments

Level C-D: **Recommended**

[Ellipsis] Do not use unspecified function arguments (ellipsis notation).
Exception: N/A

The best known function which uses unspecified arguments is printf(). The use of such functions is not advised since the strong type checking provided by C++ is thereby avoided. Some of the possibilities provided by unspecified function arguments can be attained by overloading functions and by using default arguments.

Level C-D: **Recommended**

[RefOrPtrArg] If a function is to store the object that is passed in via an argument, and thereby take “ownership” of the object, the argument should be a pointer type. Use reference arguments in other cases. Passing arguments by value of type class is not recommended.
Exception: N/A

Member functions that store pointers that have been provided as arguments should document this by declaring the argument as a pointer instead of as a reference. This communicates transfer of ownership of the object. This provides a semantic difference between the use of pointers and references. The documentation of the function should also explicitly describe that the ownership of the object pointed to is being transferred.

One difference between references and pointers is that there is no null-reference in the language, whereas there is a null-pointer. This means that an object must have been allocated before passing it to a function. Therefore, it is not necessary to test the existence of the object within the function.

Level C-D: **Recommended**

[ConstRef] Use constant references (const &) instead of call-by-value, unless using a predefined data type or a pointer.
Exception: N/A

C++ invokes functions according to call-by-value. This means that the function arguments are copied to the stack via invocations of copy constructors, which, for large objects, reduces performance. In addition, destructors will be invoked when exiting the function. const & arguments mean that only a reference to the object in question is placed on the stack (call-by-reference) and that the object's state (its instance variables) cannot be modified. (At least some const member functions are necessary for such objects to be at all useful).

Level C-D: **Recommended**

[ExternalRef] For any class that requires a reference to an external object a public method should be provided to set this reference instead of passing the reference as a parameter in the constructor argument list.
Exception: N/A

This eliminates the construction order dependency for objects that require references to each other.
Any object that constructs multiple objects requiring references to each other should call the appropriate set methods in the owned objects after construction to resolve these references.

Example 40: Different mechanisms for passing arguments

A copy of the argument is created on the stack. The copy constructor is called on entry, and the destructor is called at exit from the function. This may lead to very inefficient code.

```
void foo1(String s);
String a; foo1(a); // call-by-value
```

The actual argument is used by the function and it can be modified by the function.

```
void foo2(String& s);
String b;
foo2(b); // call-by-reference
```

The actual argument is used by the function but it cannot be modified by the function.

```
void foo3(const String& s);
String c;
foo3 ( c ); // call-by-constant-reference
```

A pointer to the actual argument is used by the function.

```
void foo4(const String* s);
String d;
foo4(&d); // call-by-pointer-to-constant-data.
```

4.3.2 Function Overloading

Level C-D: **Recommended**

[FuncOverload] When overloading functions, all variations should have the same semantics (be used for the same purpose).
Exception: N/A

Overloading of functions can be a powerful tool for creating a family of related functions that only differ as to the type of data provided as arguments. If not used properly (such as using functions with the same name for different purposes), they can, however, cause considerable confusion.

Level C-D: **Prohibited**

[NonVirtReDef] Non-virtual methods of a base class shall not be redefined in a sub-class.
Exception: No exception.

So the C++ compiler allows such a redefinition, this could lead to calling different methods for the same object, in function of the pointer type used.

If such a redefinition appears to be needed, a semantic choice shall be made between:
(re-)designing the base class so that the method may be redefined by an inheriting class (=> virtual method, which may be redefined by polymorphism) designing the inheriting class so that it complies with the existing method of the base class)=> non-virtual method, which should not be redefined.

Example 41: Example of the proper usage of function overloading

Used like this:

```
String x = "abc123";
String y = "ab";

class String
{
public:
      …
      int contains(const char c); int i = x.contains('b');
      int contains(const char* cs); int j = x.contains(« bc1 »);
      int contains(const String& s); int k = x.contains(y);
      …
};
```

4.3.3 Return Types and Values

Level C-D: **Prohibited**

[ReturnLocal] A public function shall never return a reference or a pointer to a local variable.
Exception: No exceptions.

If a function returns a reference or a pointer to a local variable, the memory to which it refers will already have been deallocated, when this reference or pointer is used. The compiler may or may not give a warning for this.

Level C: **Mandatory**
Level D: **Unrestricted**

[ReturnValueType] Every function returned value shall have the same type as the declared function type. A function which does not return a value shall be declared type void.
Exception: No exceptions.

4.3.4 Inline Functions

Level C-D: **Optional**

[Accessors] Access functions should be declared inline.
Exception: Level C code requires structural coverage. If the structural coverage measurement toolset being used does not support measuring coverage of inline functions, declaring functions as inline is prohibited.

The normal reason for declaring a function inline is to improve performance. Correct usage of inline functions may also lead to reduced size of code, by not having the function call protocol being executed.
Modern structural coverage tools recognize inline functions and correctly report their usage / coverage. If the structural coverage tool to be used is not able to report structural coverage on inline functions, then inline functions should not be used.

4.3.5 Macro Functions

Level C-D: **Prohibited**

[Macros] Preprocessor macros shall not be used to perform functional operations. Instead use inline functions.
Exception: Any macros defined in legacy code or COTS software.

Macro functions suffer from the lack of type safety, and therefore should not be used.

Inline functions have the advantage of often being faster to execute than ordinary functions. The disadvantage in their use is that the implementation becomes more exposed, since the definition of an inline function shall be placed in an include file for the class, while the definition of an ordinary function may be placed in its own separate file. Virtual methods and class constructors/destructors should not be inlined.

A result of this is that a change in the implementation of an inline function can require comprehensive re-compiling when the include file is changed. This is true for traditional file-based programming environments that use such mechanisms as make for compilation.

Example 42: Inline functions are better than macros

Example of problems with #define "functions"

```
#define SQUARE(x) (x * x)
int a = 2;
int b = SQUARE(a++); b = (2 * 3) = 6
```

Inline functions are safer and easier to use than macros if you need an ordinary function that would have been unacceptable for efficiency reasons. They are also easier to convert to ordinary functions later on.

```
inline int square(int x)
{
      return (x * x);
};
int c = 2;
int d = square(c++);
d = ( 2 * 2 ) = 4;
```

4.3.6 Temporary Objects

Level C-D: **Recommended**

[CompilerTemp] Minimize the number of temporary objects that are created as return values from functions or as arguments to functions.
Exception: N/A

[TempLife] Do not write code that is dependent on the lifetime of a temporary object.
Exception: N/A

Temporary objects are often created when objects are returned from functions or when objects are given as arguments to functions. In either case, a constructor for the object is first invoked; later, a destructor is invoked. Large temporary objects make for inefficient code. In some cases, errors are introduced when temporary objects are created. It is important to keep this in mind when writing code. It is especially inappropriate to have pointers to temporary objects, since the lifetime of a temporary object is undefined.

Temporary objects are often created in C++, such as when functions return a value. Difficult errors may arise when there are pointers in temporary objects. Since the language does not define the life expectancy of temporary objects, it is never certain that pointers to them are valid when they are used. In addition if the temporary has pointers to local data in the function that was just exited, then the data will be invalid because it is out of scope.

One way of avoiding this problem is to make sure that temporary objects are not created. This method, however, is limited by the expressive power of the language and is not generally recommended.

The C++ standard may someday provide an solution to this problem. In any case, it is a subject for lively discussions in the standardization committee.

Example 43: Difficult error in a string class which lacks output operator

```
class String
{
public:
      // Conversion operator to
      // const char* friend String
      // operator+(const String& left, const String& right); operator const
      char*() const;
      …
};
String a = "This phrase may get ";
String b = "lost!";
```

The addition of a and b generates a new temporary String object. After it is converted to a char* by the conversion operator, it is no longer needed and may be deallocated. This means that characters that are already deallocated are printed to cout -> DANGEROUS.

```
cout << a + b;.
```

4.3.7 Assignment Statement Function Arguments

Level C-D: **Prohibited**

[AssignStat] The assignment statement shall not be used as a parameter in a function call.
Exception: No exceptions.

4.3.8 Recursive Functions

Level C-D: **Restricted**

[RecFunct] If recursive functions are used then design should contain explicit safeguards to avoid stack over run from unlimited recursion.
Exception: N/A

4.3.9 Re-entrant Functions

Level C-D: **Restricted**

[Re-EntFunct] The use of re-entrant functions should be directly traceable to explicit software requirements, which are in turn derived from system requirements. Re-entrant functions should not assign values to global variables.
Exception: N/A

4.4 Constants

Level C-D: **Recommended**

[Constants] Constants should be defined using `const` or `enum`.
Exception: N/A

If a constant has been defined using `const` , the name and value of the constant is recognized in debuggers. If the constant is represented by an expression, this expression may be evaluated differently for different instantiations, depending on the scope of the name. .Constants whose value is computed at run-time should be avoided, even when the

value is computed once and never again changed. Do not convert const objects to non-const or cast away the const-ness of an object.

Example 44: Constants

It's typesafe and often easier to debug. A common exception to this is when defining class constants, in which case an enum is normally the idiom of choice:

```
class A
{
      public:
            enum { MaxLength = 100 };
      private:
            char buffer[MaxLength];
}
class GEM_Object : public Cobject
{
      public:
            enum HIT_RESULT { NO_HIT,
                              HIT,
                              HIT_DONT_CARE}
}
```

Example 45: Different ways of declaring constants.

Constants using macros

```
#define BUFSIZE 7 //No type checking
```

Constants using const

```
const int bufSize = 7; // Type checking takes place
```

Constants using enums

```
enum SIZE { BufSize = 7 }; // Type checking takes place
```

Example 46: Declaration of const defined in another file

```
extern const char constantCharacter;
extern const String fileName;.
```

4.4.1 Use of the Keyword const

Using const objects can more accurately convey the meaning and intent of your code. The result is classes that are easier for others to use. In addition, the number of potential errors is reduced (those due to modification of an object, either inadvertently on the part of the class implementer, or intentionally on the part of the class implementer but unknown to the class user). This can often mean easier testing, debugging, and maintenance. C++ is designed such that the compiler is your ally in enforcing constness of objects once they are declared that way; it is an extremely useful feature of the language.

4.4.1.1 const Arguments

Level C-D: **Mandatory**

[ConstArgs] Whenever an argument is being passed by reference or pointer for efficiency reasons only, it shall be passed as a constant.

Exception: No exception.

Example 47: const Arguments

```
virtual BOOL compatibleDrop( const GEM_Object*,
const GEM_CPoint&) const;
```

Here, we pass arguments by pointer and by reference not because compatibleDrop() is expected to alter the contents of either one, but because it is more efficient than calling copy constructors to pass by value. The argument to a copy constructor and to an assignment operator should be a const reference. The const at the end of the declaration is discussed in the following section.

4.4.1.2 const member Function

Level C-D: **Recommended**

[ConstMF] A member function that should not affect the state of an object (its instance variables) is to be declared const.
Exception: N/A

[ExtConstMF] If the behaviour of an object is dependent on data outside the object, this data is not to be modified by const member functions.
Exception: N/A

Member functions declared as const may not modify member data and are the only functions that may be invoked on a const object. (Such an object is clearly unusable without const methods). A const declaration is an excellent insurance that objects will not be modified (mutated) when they should not be. A great advantage that is provided by C++ is the ability to overload functions with respect to their const-ness. (Two member functions may have the same name where one is const and the other is not).

Example 48: A const-declared access functions to internal data in a class

```
class SpecialAccount : public Account
{
public:
      int InsertMoney();
      int getAmountOfMoney(); No - Forbids ANY constant object to access the amount of money.
      int GetAmountOfMoney() const; Better
…
private:
      int moneyAmount;
};
```

Example 49: Overloading an operator/function with respect to const-ness

```
#include <iostream.h>
#include <string.h>

static unsigned const cSize = 1024;
class InternalData {};

class Buffer
{
public:
      Buffer(char* cp);
      // A. non-const member functions: result is an lvalue
      char& operator[](unsigned index) { return buffer[index]; }
```

```
      InternalData& Get() { return data; }
      // B. const member functions: result is not an lvalue
      char operator[](unsigned index) const { return buffer[index]; }
      const InternalData& Get() const { return data; }
private:
      char buffer[cSize];
      InternalData data;
};
inline Buffer::Buffer(char* cp)
{
      strncpy(buffer, cp, sizeof(buffer));
}
main()
{
      const Buffer cfoo = "peter"; This is a constant buffer
      Buffer foo = "mary"; This buffer can change
      foo[2] = 'c'; calls char& Buffer::operator[](unsigned)
      cfoo[2] = 'c'; //ERROR: cfoo[2] is not an lvalue.
      // cfoo[2] means that Buffer::operator[](unsigned) const is called.
      cout << cfoo[2] << ":" << foo[2] << endl; OK. Only rvalues are needed
}.
```

Declare a pointer or reference argument, passed to a function, as const if the function does not change the object bound to it. An advantage of const-declared parameters is that the compiler will actually give you an error if you modify such a parameter by mistake, thus helping you to avoid bugs in the implementation. Do not let const member functions change the state of the program. A const member function promises not to change any of the data members of the object. Usually this is not enough. It should be possible to call a const member function any number of times without affecting the state of the complete program. It is therefore important that a const member function refrains from changing static data members, global data, or other objects to which the object has a pointer or reference.

A member function that has not been declared as const may not be invoked on an object that has been declared as const. Getters should almost always be const, but there are many other types of functions that should also be declared as const.

Example 50: const Member Functions

```
Finding the instance doesn't change the container
GEM_OBJECT*GEM_Object::find Object(const GEM_Rect&,GEM_Object::OBJECT_TYPE)
const;

Tells the class user two things: first, that calling findObject() will not modify
the instance of GEM_Object that (s)he has in hand; second, that the following
sort of thing is valid:
GEM_ObjectList MyRLLClass::findAllObjectsInRect(const GEM_Rect& rect,
                  const GEM_Object& searchObject )
{
      GEM_ObjectList list;
      GEM_Object* pObject;
      if (pObject = searchObject.findObject(rect, GEM_Object::BOX ) )
            list.addTail( pObject );
      if (pObject = searchObject.findObject(rect, GEM_Object::SLIDER ) )
            list.addTail( pObject );
      …
      return list;
}
```

If findObject() had not been declared 'const', the programmer would have had to 'cast away the const':
p_Object = ((GEM_Object)searchObject).findObject(bounding, GEM_Object::BOX);
This is hardly ever desirable.

4.4.1.3 const Function Return Values

Level C-D: **Restricted**

[ConstRetVal] Whenever a reference to internal data is to be passed back, it should be declared as a const. Whenever a pointer to internal data is to be passed back, the data to which it points should be declared as constant.

Exception: N/A

Example 51: const Function Return Values

In MFC's const char* cast:

```
CString::operator const char*() const { ...  };
or when a reference to internal data is being passed back for efficiency
concerns:

class A
{
public:
   const CString& getData() const {return data };
private:
   CString data;
}
```

4.4.1.4 const Constants

Level C-D: **Recommended**

[MagicNum] Avoid the use of numeric values in code; use symbolic values instead.

Exception: Certain numerical values have a well-established and clear meaning in a program.

These may be used directly in code without being considered to be "Magic". Such as MS-DOS interrupt function numbers. Numerical values in code ("Magic Numbers") should be viewed with suspicion. They can be the cause of difficult problems if and when it becomes necessary to change a value. A large amount of code can be dependent on such a value never changing, the value can be used at a number of places in the code (it may be difficult to locate all of them), and values as such are rather anonymous (it may be that every '**2**' in the code should not be changed to a '**3**').

From the point of view of portability, absolute values may be the cause of more subtle problems. The type of a numeric value is dependent on the implementation. Normally, the type of a numeric value is defined as the smallest type that can contain the value.

Const member functions should not change the state of the program. A const member function must not change any of the data members of the object. It should be possible to call a const member function any number of times without affecting the state of the complete program. It is therefore important that a const member function refrains from changing static data members, global data, or other objects to which the object has a pointer or reference.

4.5 Variables

Level C-D: **Recommended**

[VarScope] Variables are to be declared with the smallest possible scope.

Exception: Built-in types, int, etc. are exceptions.

A variable ought to be declared with the smallest possible scope to improve the readability of the code and so that variables are not unnecessarily allocated. When a variable that is declared at the beginning of a function is used somewhere in the code, it is not easy to directly see the type of the variable. In addition, there is a risk that such a variable is inadvertently hidden if a local variable, having the same name, is declared in an internal block.

Many local variables are only used in special cases, which seldom occur. If a variable is declared at the outer level, memory will be allocated even if it is not used. In addition, when variables are initialized upon declaration, more efficient code is obtained than if values are assigned when the variable is used.

Level C-D: **Recommended**

[VarDecl] Each variable is to be declared in a separate declaration statement.
Exception: Closely related variables, such as x and y when used as a coordinate pair may be grouped together.

Level C-D: **Recommended**

[VarInit] Every variable that is declared is to be given a value before it is used.
Exception: N/A

A variable should always be initialized before use. Normally, the compiler gives a warning if a variable is undefined. It is then sufficient to take care of such cases. Instances of a class are usually initialized even if no arguments are provided in the declaration (the default constructor is invoked). To declare a variable that has been initialized in another file, the keyword extern is always used.

Level C-D: **Recommended**

[InitVsAssign] If possible, always use initialization instead of assignment.
Exception: In certain special cases, a variable is assigned the value of a complicated expression; it may then be unnecessary to give the variable an initial value.

By always initializing variables, instead of assigning values to them before they are first used, the code is made more efficient since no temporary objects are created for the initialization. For objects having large amounts of data, this can result in significantly faster code.

Example 52: Initialization instead of Assignment

Do not do this

```
int i;
// 1021 lines of code
i = 10;
int i = 10; //Better.
```

Example 53: Initialization instead of Assignment (classes)

```
class Special     // Array of this class is used to initialize
{                 // MyClass::complicated
      public:
            // Default constructor
            Special();
            int isValid() const;
            int value() const;
};
const int MaxSpecialArray = 1066;
Special specialInit[MaxSpecialArray];

class MyClass
{
      public:
            // Constructor
            MyClass(const char* init);
            …
```

```
        private:
                String privateString;
                int complicated;
};
```

Do not do this. Inefficient code. Empty constructor and assignment operator called for privateString

```
MyClass::MyClass(const char* init)
{
        privateString = init;
        // Special case - complicated expression
        for (int i = 0; i < MaxSpecialArray; i++)        //  No You should enclose "for" loops
                                                         //    in braces. See Rule EmptyFlow
                if (specialInit[i].isValid())
                {
                complicated = specialInit[i].value();
                break;
                }
}

// Better
MyClass::MyClass(const char* init) : privateString(init)
{
        // Special case - complicated expression
        for (int i = 0; i < MaxSpecialArray; i++)        //No. You should enclose "for" loops
                                                         // in braces. See Rule EmptyFlow.
                if (specialInit[i].isValid())
                {
                        complicated = specialInit[i].value();
                        break;
                }
}
```

Level C-D: **Mandatory**

[ImpTest] The code shall not use an implicit test for 0 other than for Boolean variables and pointers.
Exception: Float variable is allowed

Example 54: Using 0 with a float value

```
if (scale == 0.0 || scale == 1.0) //Scale is of float type
```

Example 55: Implicit test for 0 should be avoided

```
if (nLines != 0)  // NOT: if (nLines)
if (value != 0.0)  // NOT: if (value)
```

It is not necessarily defined by the C++ standard that ints and floats 0 are implemented as binary 0. Also, by using explicit test the statement give immediate clue of the type being tested. It is common also to suggest that pointers shouldn't test implicit for 0 either, i.e. if (line == 0) instead of if (line). The latter is regarded so common in C/C++ however that it can be used.

4.6 Pointers and References

4.6.1 Pointer Type Definitions

Level C-D: **Mandatory**

[PtrDeclaration] A Pointer declaration shall explicitly declare the data type, which shall be the same as every variable to which it points.
Exception: No exception

Code designers or maintainers should not have to make assumptions about the type of the variable, which is being set in the code.

4.6.2 Pointer Indirection

Level C-D: **Restricted**

[PtrIndirection] If the level of indirection exceeds one in the use of pointers, then verification of the code will include test cases specially documented to assure the robustness of the design.
Exception: N/A

When the level of indirection increases then the code becomes difficult to understand, verify and maintain

4.6.3 Pointers to Automatic Storage

Level C: **Prohibited**
Level D: **Restricted**

[PtrAutoVariable] Static pointers shall not be assigned the address of a storage-class auto variable.
Exception: N/A

If a static pointer is set to the address of a storage-class auto variable, the pointer shall be set to a storage-class static address or NULL before the function executes a return.

Level C-D: **Recommended**

[Ptr2Ptr] Pointers to pointers should be avoided whenever possible.
Exception: N/A

Pointers to pointers normally should not be used. Instead, a class should be declared, which has a member variable of the pointer type. This improves the readability of the code and encourages data abstraction. By improving the readability of code, the probability of failure is reduced. One exception to this rule is represented by functions that provide interfaces to other languages (such as C). These are likely to only allow pre-defined data types to be used as arguments in the interface, in which case pointers to pointers are needed. Another example is the second argument to the main function, which shall have the type char*[]. This is equivalent to char**.

A function which changes the value of a pointer that is provided as an argument, should declare the argument as having the type reference to pointer (e.g. char*&).

Example 56: Pointers to pointers are often unnecessary

This example is taken, in part, from: [2].

```
#include <iostream.h>
void printM_RowCol(int** m, int dim1, int dim2)
{
```

```
for (int row = 0; row < dim1; ++row)
      {
      for (int col = 0; col < dim2; ++col)
            cout << " " << ((int*)m)[row * dim2 + col]; cout << endl;
      }
}
```

Could be written as:

```
class Int_Matrix
{
public:
      Int_Matrix(int dim1, int dim2);
      int value(int,int) const;
      int dim1() const;
      int dim2() const;
      …
};

void printM_RowCol(Int_Matrix m)
{
      for (int row = 0; row < m.dim1(); ++row)
      {
            for (int col = 0; col < m.dim2(); ++col)
                  cout << " " << m.value(row, col);
            cout << endl;
      }
}
```

Level C-D: **Recommended**

[Ptr2Func] Use a typedef to simplify program syntax when declaring function pointers.
Exception: N/A

A typedef is a good way of making code more easily maintainable and portable. Another reason to use typedef is that the readability of the code is improved. If pointers to functions are used, the resulting code can be almost unreadable. By making a type declaration for the function type, this is avoided.
Function pointers can be used as ordinary functions; they do not need to be dereferenced.

Example 57: Complicated declarations

func1 is a function: int -> (function : const char* -> int) (i.e. a function having one argument of type int and returning a pointer to a function having one argument of type const char* and returning an int.

```
int (*func1(int))(const char*);
```

func1 of the same type as func2

```
typedef int FTYPE(const char*);
FTYPE* func2(int);
int (*(*func1p)(int))(const char*) = func2;
```

Realistic example from signal.h

```
void (*signal(int,void (*)(int)))(int);
```

Example 58: Syntax simplification of function pointers using a typedef

```
#include <math.h>
```

Ordinary messy way of declaring pointers to functions:

```
double (*mathFunc)(double) = sqrt;
```

With a typedef, life is filled with happiness (Chinese proverb):

```
typedef double MathFuncType(double);
MathFuncType* mathFunc = sqrt;
main()
{
      You can invoke the function in an easy or complicated way
      double returnValue1 = mathFunc( 23.0 );  Easy way
      double returnValue2 = (*mathFunc)(23.0);  No. Correct, but
                                                complicated
}.
```

Level C-D: **Recommended**

[Ptr2Type] A typedef should not be used on structures, unions, enumeration, or classes to support forward references when declaring pointers.

Exception: A typedef must be used within a class or namespace due to limitations in Watcom C++ compiler.

Avoiding the use of the typedef with struct, union, enum, and class allows forward references on pointers. There should only be a tag reference. This represents the evolution of the C++ language and allows for untangling of header files dependencies. In addition, the Microsoft Studio Debugger needs the tag to correctly resolve the name. When the tag is within a namespace or class, then the type and the tag should be the same to reduce the number of names in the system. Note that tags and types may be used interchangeably after they are defined. Note that the Watcom compiler will not export tags from within a class or namespace.

Example 59: Structure reference

forward reference (struct is required since tag is not defined)

```
struct realtime* RealTime;
struct realtime
{
      unsigned char hours;
      unsigned char minutes;
      unsigned char seconds;
};
```

struct not required since tag is defined

```
realtime* myRealTime;
```

Example 60: Class reference

forward reference to class (class is required since tag is not defined)

```
class Time* myTime;
class Time
{
public:
      typedef struct realtime
      {
            unsigned char hours;
            unsigned char minutes;
            unsigned char seconds;
      } realtime;
};
```

Note that forward references to inside a class are illegal, therefore the class must be defined first

```
Time::realtime* RealTime;
```

Do not access a pointer or reference to a deleted object. A pointer that has been used as argument to a delete expression should not be used again unless you have given it a new value, because the language does not define what should happen if you access a deleted object. You could assign the pointer to 0 or a new valid object. Otherwise you get a "dangling" pointer. Do not cast a pointer to a shorter quantity to a pointer to a longer quantity. Certain types have alignment requirements, which are requirements about the address of objects. For example, some architectures require that objects of a certain size start at an even address. It is a fatal error if a pointer to an object of that size points to an odd address. For example, you might have a char pointer and want to convert to an int pointer. If the pointer points to an address that it is illegal for an int, dereferencing the int pointer creates a runtime error.

4.6.4 Null Pointer

Level C-D: **Recommended**

[NullPtr] Pointers should be compared to the macro NULL. The project should define NULL to be equal to the value 0.
Exception: N/A

NULL is more readable and is the common term communicated and understood by software engineers

4.7 Type Conversions

Level C-D: **Mandatory**

[CastVBase] Conversion of a pointer to an object of a base class to be a pointer to an object of a derived class shall only be allowed in cases where it is known for certain the type of the derived class.
Exception: N/A

Without knowledge of a derived class' type, it is not allowed to cast a base class pointer to be the type of a derived class pointer.

Example 61: Downcasting to uncertain derived class is unsafe

```
class  Base
{
 public:
    virtual void Run() {printf("Do base class work\n");}
};

class  Derived : public Base
{
  int m_st;
  public:
       virtual void Run();
      void SetState(int d);
};

void Derived::Run()
{
      printf("Run derived class process\n");
}
void Derived::SetState(int d)
{
      m_st = d;
}

int main()
{
```

```
        Base *bP = new Base();
        Derived *dP = (Derived *) bP; // #1 Unsafe type cast

       return 0;
}
```

Type cast #1 is not safe because it assigns the address of a base-class object to a derived class pointer. So, the code would expect the base-class object to have derived class properties such as SetState() method, and that is false. Also, derived object, for example, has a member classes that a base object is lacking.

Level C-D: **Recommended**

[LongtoShort] Type conversions made from a longer to shorter data types should explicitly include documentation of what bits will be lost along with their definition.
Exception: N/A

This clarifies the loss of data from truncation that would occur normally in an easy-to overlook, implicit cast operation.

Use explicit rather then implicit type conversion. It is important to use explicit conversion, as implicit conversion is almost always bad and makes the code less robust, less portable and less readable. Never rely on implicit type conversion.

All implicit conversions should be checked for potential loss of precision or loss of data. Specifically, the following should be justified:

- From integer types to the floating point types
- From a floating point type to an integer type
- From a more precise numeric type to a less precise version of the same numeric type (e.g., long to short, double to float, etc.)

Implicit conversions between logically unrelated types should be justified. Types are logically unrelated when one does not define a set of operations that is a subset of the other. Not using the explicit keyword should be justified.

4.8 Expressions

Level C-D: **Recommended**

[SubExpr] Sub expressions of large expressions should provide natural breaks for formatting the expression.
Exception: N/A

It is common for large expressions to exceed the page (screen) width. These expressions should be reformatted with the expression being broken into two or more lines at a sub expression level. The operator that separates sub expressions should be placed at the end of the preceding expression.

4.9 Memory Allocation

Level C-D: **Mandatory**

[DelArray] The code shall always provide empty brackets ("[]") for delete when deallocating arrays.
Exception: No exceptions.

If an array 'a' having a type 'T' is allocated, it is important to invoke delete in the correct way. Only writing 'delete a'; will result in the destructor being invoked only for the first object of type '**T**'. By writing 'delete [m]' a; where 'm' is an integer which is greater than the number of objects allocated earlier, the destructor for 'T' will be invoked for memory that may not represent objects of type 'T'. The easiest way to do this correctly is to write 'delete [] a'; since the destructor will then be invoked only for those objects which have been allocated earlier.

[NewDelDef] The implementation of new and delete operators shall be redefined by using manually handled pools, instead of standard compiler implementation. (This is a certification only.)

Exception: No exceptions.

[NoMallocFree] The <<C standard>> memory management functions malloc(), calloc(), realloc(), free(), . . . shall not be used in C++ programs. The C++ standard new and delete functions shall be used.
Exception: If strictly necessary to use these functions, make sure neither to use C standard allocators (alloc()) on objects having a constructor, nor to use free() on a pointer allocated via new, nor to use delete on a pointer obtained via a C standard allocator (alloc()).

Example 62: Right and wrong ways to invoke delete for arrays with destructors

```
int n = 7;
T* myT = new T[n]; // T is a type with defined constructors and destructors
delete myT; // No. Destructor only called for first object in array T
delete [10] myT; // No. Destructor called on memory past the end of array T
delete [] myT; // OK, and always safe.
```

4.9.1 Dynamic Allocation; New, Delete

Level C: **Restricted**
Level D: **Unrestricted**

[DynMemAlloc] Functions should not use dynamic memory allocation, and no functions should use the legacy C language memory functions, malloc, realloc and free. The C++ functions new and delete should be used in applications only during initialization to allocate and free storage.
Exception: N/A

The new functions are a distinct improvement to the old malloc and free. These include automatic type conversion of the pointer returned by new and automatic size computation to allocate arrays of a data type. Dynamic Allocation is still subject to many types of failure, and the user must correctly include deletes in the design to match up with the use of new to avoid memory leaks.

4.9.2 Global Data Initialization

Level C-D: **Mandatory**

[GlobalDataInit] Global Data shall be initialized before use.
Exception: No exceptions.

This ensures that in the case of a cold restart of the system, the global data will be correctly reinitialized.

4.9.3 Resource Exhaustion; Set_New_Handler

Level C: **Prohibited**
Level D: **Unrestricted**

[ResExhaustion] A call to new, which encounters memory exhaustion, shall cause the system to throw an exception. The code shall not invoke exception handling for memory exhaustion with the set_new_handler function.
Exception: No exceptions.

In keeping with the exception handling policy and dynamic memory allocation, memory exhaustion will not occur in a system. In the (extremely unlikely) event that a latent design error causes memory exhaustion, then an exception will halt the system.

4.10 Memory Usage

4.10.1 Type Casting

Level C-D: **Mandatory**

[TypeCast] Type casting shall be explicitly coded with use of the C++ keywords or the standard C cast operator, and shall not be left as an operation only implied by the code (e.g., an assignment statement).
Exception: No exceptions.

The requirement makes each cast identifiable to searches, and makes it necessary for the programmer to be aware of the data conversion, for example, whether or not the conversion is portable or not.

4.10.2 Auto Variables

Level C-D: **Prohibited**

[AutoVar] A function shall set the values of *storage class* auto variables before the variables are used. A function shall not return the address of an automatic variable.
Exception: No exceptions.

An automatic variable, which is used before it is set, will pick up whatever random memory contents are already there. A function, which acquires such a random value, might then inadvertently pass verification test by chance.

4.10.3 Run Time Type Identification (RTTI)

Level C-D: **Prohibited**

[RTTI] The dynamic_cast keyword shall not be used to accomplish Run Time Type Identification.
Exception: No exceptions.

If you need runtime typing, you can achieve a similar result by adding a classOf() virtual member function to the base class of your hierarchy and overriding that member function in each subclass. If classOf() returns a unique value for each class in the hierarchy, you'll be able to do type comparisons at runtime.

4.10.4 Assignment Type Consistency

Level C-D: **Mandatory**

[AssignTypeCons] Assignment statements shall contain type-consistent variables, i.e., mixed-mode expressions are not allowed. Any type differences shall be addressed by casting (explicit type conversion) to the correct type.
Exception: No exceptions.

An assignment with mismatching variable types, which have different sizes, can drop or pick up data. This can be missed in testing because difference in type can pick up or drop values with seemingly random effect.

4.10.5 Type Bool

Level C-D: **Mandatory**

[Bool] The C++ language data type, bool shall be used for all Boolean functions and operations.
Exception: No exceptions.

There are different implementations of Boolean operations in legacy C code, which are not all mutually compatible or portable. It is recommended that if such legacy C code is included in a C++ development that the text be scanned for Boolean type data definitions and these be converted to type bool. Use of built in types is inherently safer.

4.10.6 Constructors / Destructors

Level C-D: **Mandatory**

[ConstDest] Design of a class shall include destructor definitions for appropriate cleanup when the class is closed.
Exception: No exceptions.

4.11 Control Flow

4.11.1 Break In Switch

Level C: **Mandatory**
Level D: **Unrestricted**

[Switch] Each case in a switch, which contains any executable code, shall be terminated with break. It is too easy to lose track of the correct program flow in code maintenance. However, it is acceptable to code a set of case statements which all execute the same code.
Exception: No exceptions.

Example 63: Allowable Construction

```
case 'x':
case 'y':
case 'z':
{ ... code statements ...
break; }
```

Example 64: Unacceptable Construction:

```
case 'x':
myloc = there + 3;
case 'y':
yourloc = here + 2;
case 'z':
{ ... code statements ...
break; }
```

4.11.2 Default Case

Level C: **Mandatory**
Level D: **Unrestricted**

[SwitchDefault] All Default statements shall be left empty, contain an assert, print an error to the trace port, or contain executable code.
Exception: No exceptions.

Example 65: Allowable Construction

```
case 'x':
case 'y':
case 'z':
default:
{ printf("error 12: %i\n", temp);
  assert(temp > 0);
break; }
```

Example 66: Allowable Construction

```
case 'x':
case 'y':
case 'z':
default:
   break;
```

4.11.3 Exception Handling

Level C: **Prohibited**
Level D: **Unrestricted**

[ExeHandl] Exception Handling shall not be used. Exceptions are another C++ feature which is not very widely implemented. Unfortunately, there is no good workaround that produces similar functionality.
Exception: No exceptions.

4.12 Standard Libraries

4.12.1 Commercial Off-the-Shelf Software (COTS)

Level C-D: **Restricted**

[COTS] If the **C++** Standard Template Library (STL) routines are used, which are delivered as part of a compiler package, or some other source external to the supplier is used, then it should comply with requirements of Software Considerations in Airborne Systems and Equipment Certification, [DO-178B, RTCA Inc., March 26, 1999], which apply to COTS for those library routines.
Exception: N/A

APPENDIX A: Definitions

abstract base class ---------- 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. In C++, a class is abstract if it includes at least one member function that is declared as pure virtual.

abstract data type ---------- A class/struct/union is said to be an abstract data type if it does not have any public or protected data members.

Accessor ---------- An accessor is a function that returns the value of a data member.

C++ARM ---------- The C++ Annotated Reference Manual, Stroustrup and Ellis, Reference [2]. This book along with the ANSI/ISO C standard serves as the base documents for the ANSI/ISO standardization effort for C++.

catch clause---------- 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.

class ---------- A class is a user-defined data type that consists of data elements and functions that 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. The default access control of members is private.

compilation unit ---------- A compilation unit is the source code (after preprocessing) that is submitted to a compiler for compilation (including syntax checking).This is normally a single source file with its included files. constant member function A constant member function is a function that may not modify data members.

constructor ---------- A constructor is a function that creates and initializes an object. Its name is the same as the class. It creates a valid data object from raw memory. The compiler will generate default and copy constructors if none are provided in the implementation.

copy constructor---------- A copy constructor is a constructor that can be called with a single argument of a reference to an object that has the same type as the object to be initialized. It is used to make a copy of an object. The default copy constructor that a compiler generates will perform a bitwise copy of the original object.

default constructor---------- A *default constructor* is a constructor that needs no arguments.

destructor---------- A destructor is a function that destroys an object, performing any cleanup that is required after an object is out of scope.

enumeration type---------- An enumeration type is an explicitly declared set of symbolic integral constants. In C++ it is declared as an enum.

exception An exception is a run-time program/system event that is detected in a function or member function. Exception handling provides for the uniform management of exceptions. When an exception is detected, it is thrown (using a throw expression) to the exception handler.

forwarding function ---------- A forwarding function is a function that does nothing more than call another function.

hidden ---------- A hidden member function is a member function in a base class that is re-defined in a derived class. Such a member function may be declared virtual if the derived function has a different signature than the base function. The function may be declared non-virtual and then the derived class function totally hides the base class function.

identifier ---------- An identifier is a name that is used to refer to a variable, constant, function or type in C++.

implementation file ---------- An implementation file is a file that contains the source code for a class' member functions. Its file extension is .cpp.

iterator ---------- An iterator is an object that, when invoked, returns the next object from a collection of objects.

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

member ---------- A member is a function or data that is part of a class.

overloaded function name ------- An overloaded function name is a name that is used for two or more functions or member functions having different signatures. Functions cannot be overloaded

however solely by having different return types.

overridden ---------- An overridden member function is a member function in a base class that is re-defined in a derived class. In C++, this member function is declared virtual. See also hidden and overloaded function name.

pre-defined data type ---------- A pre-defined data type is a type that is defined in the language itself, such as int.

protected members ---------- Protected members of a class are member data and member functions that are accessible by derived classes. These members are not accessible by non-derived classes and other external functions.

public members ---------- Public members of a class are member data and member functions that are accessible everywhere by specifying an instance of the class and the member name.

pure virtual function ---------- A pure virtual function is a member function for which no definition is provided. Pure virtual functions are specified in abstract base classes and shall be defined (overridden) in derived classes.

reference ---------- 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.

scope ---------- The scope of a name refers to the region of code, blocks, functions, modules in which the name is visible and can be used.

signature ---------- The signature of a function is a combination of return type and function arguments that identify a specific definition of a function.

structure ---------- A structure is a user-defined type that consists of data elements and functions that operate on that data. The default access control of members is public. See also class.

template class ---------- A template class 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.

typedef ---------- A typedef is another name for a data type, specified in C++ by using a typedef declaration. Classes and structure names are automatically typedef'd.

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

APPENDIX B: Summary

The number in parenthesis and italics following a checklist item is the page where the standard or guideline is explained in the document.

Standards

□ **Underscore** Developers shall not use identifiers which begin with one or two underscores ('_' or '__'). (9)

□ **IncludeExt** Include files shall always have the file name extensions ".h". *(11)*

□ **SourceExt** Implementation files in C++ shall always have the file name extension ".cpp". *(11)*

□ **ClassDef** An include file shall not contain more than one class definition. *(11)*

□ **NestIncludes** Every include file shall contain a mechanism that prevents multiple inclusions of the file. *(13)*

□ **IncludeForm** The directive #include "filename*.h"* shall be used for ALL include files. *(15)*

□ **AccessControl** The public, protected, and private sections of a class shall be declared in that order (the public section is declared before the protected section, which is declared before the private section). *(19)*

□ **NoPubAttr** A class shall not have public attributes. *(20)*

□ **StdClass** All classes shall define a default constructor, a copy constructor, an assignment operator (operator=()), and a destructor. *(20)*

□ **GlobalsInCtor** Global objects shall not be used in constructors and destructors. *(22)*

□ **StaticInit** It shall not be assumed that static objects are initialized in any special order. *(22)*

□ **OpAssign** An assignment operator which performs a destructive action shall be protected from performing this action on the object upon which it is operating. *(25)*

□ **Polymorphism** All public and protected functions in a base class or an intermediate class shall be declared as virtual or pure virtual. *(27)*

□ **VirtualDtor** All classes shall have a virtual destructor. *(28)*

□ **NonVirtReDef** Non-virtual methods of a base class shall not be redefined in a sub-class. *(33)*

□ **ReturnLocal** A public function shall never return a reference or a pointer to a local variable. *(34)*

□ **Comments** All comments shall be written in English. *(12)*

□ **NameAbbrev** Names shall not include ambiguous abbreviations. *(9)*

□ **ConstArgs** Whenever an argument is being passed by reference or pointer for efficiency reasons only, it shall be passed as a constant. *(37)*

□ **Copyright** All code shall be copyrighted. *(13)*

□ **DataDef** Data definitions shall be explicit, i.e. long, short, char. Do not use int or unsigned without an explicit qualifying long, short, char, etc. *(18)*

□ **DelArray** The code shall always provide empty brackets ("[]") for delete when deallocating arrays. *(47)*

□ **NoMallocFree** The <<C standard>> memory management functions malloc(), calloc(), realloc(), free(), . . . shall not be used in C++ programs. The C++ standard new and delete functions shall be used. *(48)*

□ **SwitchDefault** All Default statements shall be left empty, contain an assert, print an error to the trace port, or contain executable code. *(50)*

□ **IncludeH** When the following kinds of definitions are used (in implementation files or in other include files), they shall be included as separate include files: *(14)*

- classes that are used as base classes,
- classes that are used as member variables,
- classes that appear as return types or as argument types in function/member function prototypes.
- function prototypes for functions/member functions used in inline member functions that are defined in the file.

□ **Names** Names shall have the following properties: *(8)*

- Member variables should be prefixed with “m_” and member pointer variables with “m_p”. Following the prefix, the rest of the variable should begin with an uppercase letter. (e.g. m_VariableName or m_pPointerName).
- The names of preprocessor Macros are to be entirely uppercase with the words separated by an underscore ‘_’.

□ **RefByName** Definitions of classes that are only accessed via pointers (*) or references (&) shall not be included as include files. *(14)*

□ **NewDelDef** The implementation of new and delete operators shall be redefined by using manually handled pools, instead of standard compiler implementation. *(47)* (This is a certification only.)

□ **Bool** The C++ language data type, bool shall be used for all Boolean functions and operations. *(49)*

□ **ImpTest** The code shall not use an implicit test for 0 other than for Boolean variables and pointers. *(42)*

□ **ConstDest** Design of a class shall include destructor definitions for appropriate cleanup when the class is closed. *(50)*

□ **PtrAutoVariable** Static pointers shall not be assigned the address of a storage-class auto variable. *(43)*

□ **ResExhaustion** A call to new, which encounters memory exhaustion, shall cause the system to throw an exception. The code shall not invoke exception handling for memory exhaustion with the set_new_handler function. *(48)*

□ **BitField** Bit Fields shall have explicitly unsigned integral or enumeration types only. The bit field definition shall be platform independent. It supports both big endianess and little endianess. *(18)*

□ **Union** Any data type defined as a "union" shall be platform independent. Byte ordering shall be transparent to both endianess. *(18)*

□ **InitList** Initialization shall be preferred to assignment in constructors. *(22)*

□ **VirtDestr** Base class destructors shall be made public and virtual, or protected and non-virtual. *(22)*

□ **ExtLink** Entities shall not be defined with linkage in a header file. *(22)*

□ **VirtualInCtor** Calling virtual functions in a base constructor shall be prohibited. *(23)*

□ **PtrDeclaration** A Pointer declaration shall explicitly declare the data type, which shall be the same as every variable to which it points. *(43)*

□ **CastVBase** Conversion of a pointer to an object of a base class to be a pointer to an object of a derived class shall only be allowed in cases where it is known for certain the type of the derived class. *(46)*

□ **GlobalDataInit** Global Data shall be initialized before use. *(48)*

□ **TypeCast** Type casting shall be explicitly coded with use of the C++ keywords or the standard C cast operator, and shall not be left as an operation only implied by the code (e.g., an assignment statement). *(49)*

□ **AutoVar** A function shall set the values of *storage class* auto variables before the variables are used. A function shall not return the address of an automatic variable. *(49)*

□ **RTTI** The dynamic_cast keyword shall not be used to accomplish Run Time Type Identificatio*n.(49)*

□ **AssignTypeCons** Assignment statements shall contain type-consistent variables, i.e., mixed-mode expressions are not allowed. Any type differences shall be addressed by casting (explicit type conversion) to the correct type. *(49)*

□ **Switch** Each case in a switch, which contains any executable code, shall be terminated with break. It is too easy to lose track of the correct program flow in code maintenance. However, it is acceptable to code a set of case statements which all execute the same code.*(50)*

□ **ExeHandl** Exception Handling shall not be used.*(51)*

□ **Macros** Preprocessor macros shall not be used to perform functional operations. Instead use inline functions. *(35)*

□ **TmpCust** Templates shall be customized intentionally and explicitly.*(29)*

□ **TmpSpl** A template specialization shall be declared before its use. *(30)*

□ **NestedTemplates** Templates shall not be nested and each instance of a template shall be tested. Each instance of a template with a unique set of arguments should be tested.*(31)*

□ **ReturnValueType** Every function returned value shall have the same type as the declared function type. A function which does not return a value shall be declared type void.*(34)*

□ **AssignStat** The assignment statement shall not be used as a parameter in a function call.*(36)*

Guidelines

□ **IncludeCpp** Every implementation file is to include the relevant files that contain: *(14)*

- declarations of types and functions used in the functions that are implemented in the file.
- declarations of variables and member functions used in the functions that are implemented in the file.

□ **Paths** Do not specify relative paths in #include directives. *(15)*

□ **Standard** Every time a rule in boldface is broken, the code should be clearly documented. *(6)*

□ **Constants** Constants should be defined using `const` or `enum`. *(36)*

□ **CtorInitOrder** Do not assume that an object is initialized in any special order in constructors initialization. *(24)*

□ **ConstMF** A member function that does not affect the state of an object (its instance variables) is to be declared const. *(38)*

□ **ExtConstMF** If the behavior of an object is dependent on data outside the object, this data is not to be modified by const member functions. *(38)*

□ **FuncDef** When defining functions, the complete function definition will be on the same line including return type, function name, and any arguments. *(16)*

□ **InitVsAssign** If possible, always use initialization instead of assignment. *(41)*

□ **Presentation** Indentation should be three spaces for each level. Spaces should be used; tabs should not be included in source code. Blank lines will be included to delineate major blocks of code from each other. *(15)*

□ **VarScope** Variables are to be declared with the smallest possible scope. *(40)*

□ **VarInit** Every variable that is declared is to be given a value before it is used. *(41)*

□ **Globals** Encapsulate global functions, variables, constants, enumerated types, and typedefs in a class. *(10)*

□ **Ellipsis** Do not use unspecified function arguments (ellipsis notation). *(32)*

□ **VarDecl** Each variable is to be declared in a separate declaration statement. *(41)*

□ **IdPrefix** The identifier of every globally visible class, enumeration type, type definition, function, constant, and variable in a subsystem is to begin with a prefix that is unique for that subsystem. *(8)*

□ **PrefixSeparator** Prefixes should be separated from the name by an underscore ('_'), if it is of the same case as the first character of the name. See Exception to rule Names. *(8)*

□ **IdLength** The maximum length of an identifier is 32 characters. *(8)*

□ **NameUsage** Choose variable names that suggest the usage. *(9)*

□ **ClassImpl** A class implementation should be in one file. *(11)*

□ **ClassImpl2** A class implementation that is very large (greater than 3000 lines), should be reviewed for opportunities to introduce inheritance. *(11)*

□ **FileNames** Always give a file a name that is unique in as large a context as possible. *(11)*

□ **Indent** Blocks of code will be indented three spaces. *(16)*

□ **Braces** Braces ("{}") which enclose a block are to be placed in the same column, on separate lines directly before and after the block. *(16)*

□ **DataFormat** Data should be formatted to reflect the logical organization. *(17)*

□ **EmptyFlow** The flow control primitives if, else, while, for and do should be followed by a block, even if it is an empty block. *(17)*

□ **SimpleFlow** The expression controlled by the flow control primitives if, else, while, for and do should be placed on a new line. *(17)*

□ **AddressTypes** The dereference operator **'*'** and the address-of operator '&' should be directly connected with the type names in declarations and definitions. *(17)*

□ **AssignReturn** An assignment operator ought to return a const reference to the assigning object. *(25)*

□ **RefOrPtrArg** If a function is to store the object that is passed in via an argument, and thereby take "ownership" of the object, the argument should be a pointer type. Use reference arguments in other cases. Passing arguments by value of type class is not recommended. *(32)*

□ **ConstRef** Use constant references 'const &' instead of call-by-value, unless using a predefined data type or a pointer. *(32)*

□ **ExternalRef** For any class that requires a reference to an external object a public method should be provided to set this reference instead of passing the reference as a parameter in the constructor argument list. *(33)*

□ **FuncOverload** When overloading functions, all variations should have the same semantics (be used for the same purpose). *(33)*

□ **CompilerTemp** Minimize the number of temporary objects that are created as return values from functions or as arguments to functions. *(35)*

□ **TempLife** Do not write code which is dependent on the lifetime of a temporary object. *(35)*

□ **Ptr2Ptr** Pointers to pointers should be avoided when possible. *(43)*

□ **Ptr2Func** Use a 'typedef' to simplify program syntax when declaring function pointers. *(44)*

□ **Ptr2Type** A typedef should not be used on structures, unions, enumeration, or classes to support forward references when declaring pointers. *(45)*

□ **SubExpr** Sub expressions of large expressions should provide natural breaks for formatting the expression. *(47)*

□ **Re-EntFunct** The use of re-entrant functions should be directly traceable to explicit software requirements, which are in turn derived from system requirements. Re-entrant functions should not assign values to global variables. *(36)*

□ **NullPtr** Pointers should be compared to the macro NULL. The project should define NULL to be equal to the value 0. *(46)*

□ **LongtoShort** Type conversions made from a longer to shorter data types should explicitly include documentation of what bits will be lost along with their definition. *(47)*

□ **InitVars** Member variables should be defined and initialized in the same order. *(21)*

□ **MagicNum** Avoid the use of numeric values in code; use symbolic values instead. *(40)*

□ **PtrIndirection** If the level of indirection exceeds one in the use of pointers, then verification of the code will include test cases specially documented to assure the robustness of the design. *(43)*

□ **ClassDefFileName** An include file for a class should have a file name of the form <class identifier>.h. *(11)*

□ **Return** Always specify the return type of a function explicitly. *(20)*

□ **PublicReturn** A public member function should not return a pointer or non-const reference to member data. *(26)*

□ **PublicReturnExt** A public member function should not return a pointer or non-const reference to data outside an object, unless the object shares the data with other objects. *(26).*

□ **InheritMult** Multiple inheritance will be considered a restricted design method. *(28)*

□ **Accessors** Access functions should be declared inline. *(34)*

□ **Forwarders** Forwarding functions should be declared inline. *(34)*

□ **DynMemAlloc** Functions should not use dynamic memory allocation, and no functions should use the legacy C language memory functions, malloc, realloc and free. The C++ functions new and delete should be used in applications only during initialization to allocate and free storage. *(48)*

□ **TmpDependence** A template definition's dependence on its instantiation contexts should be minimized. *(31)*

□ **SplPtr** Specializations for pointer types should be made where appropriate. *(31)*

□ **MulDef** Avoid multiple definitions of overloaded functions in conjunction with the instantiation of a class template.*(31)*

□ **FuncTmp** Do not specialize function templates. *(30)*

□ **RecFunct** If recursive functions are used then design should contain explicit safeguards to avoid stack over run from unlimited recursion.*(36)*

□ **COTS** If the **C++** Standard Template Library (STL) routines are used, which are delivered as part of a compiler package, or some other source external to the supplier is used, then it should comply with requirements of Software Considerations in Airborne Systems and Equipment Certification, [DO-178B, RTCA Inc., March 26, 1999], which apply to COTS for those library routines. *(51)*

□ **ConstRetVal** Whenever a reference to internal data is to be passed back, it should be declared as a const. Whenever a pointer to internal data is to be passed back, the data to which it points should be declared as constant. *(40)*