Restricted

Siemens Healthcare
Business Unit Ultrasound

Title: Software C++ Coding Standards

Part Number: 10011642-QMS-001-03

Revision Data

Rev	ECO #	Change Description	Printed Name
03	660751	Updated for corrections, removal of old rules, made examples more consistent, added additional rules for C++11, C++14, and C++17, and overhauled many rules for modern C++ guidelines.	Bryan Brook/Daniel Iliescu

This document is a modified version of a document published by the Ellemtel Laboratory: graciously provided free of charge by the lab. It has been modified to meet the needs of Siemens Ultrasound Division Software Development. Pete Magsig previously modified this document at Amdahl Corporation.

The original authors of this document were Mats Henricson and Erik Nyquist of Ellemtel.

Original translation from Swedish by Joseph Supanich.

Revision History

Rev	Change Description	Printed Name	Date Made
03	Updated for corrections, removal of old rules, made examples more consistent, added additional rules for C++11, C++14, and C++17, and overhauled many rules for modern C++ guidelines.	Bryan Brook/Daniel Iliescu	12/21/2016
02	Finalized doc per formal review feedback and per Sandhya's SQA check.	Greg Sherwin/Sandhya Patel	06/10/2013
01b	Updated per review feedback and for the latest BU US document cover page.	Greg Sherwin	06/07/2013
01a	Updated for latest info and enhanced rules: 1. Added reference to C++11 ISO standards document. 2. Modified Rule 4 and added Rule 4a: constants and enumerators names begin with uppercase. 3. Added Rule 9a for naming variables of the new C++11 smart pointer types 4. Modified Rule 18 to suggest preference for using #pragma once rather than macro-based multiple include guard. 5. Added Rule 33a recommending use of enum classes (C++11) rather than traditional enums. 6. Modified Rule 91 to delete suggestion to use the Singleton design pattern as an alternative to global data because the Singleton design pattern compromises testability. 7. Modified Rule 93 to discourage use of assert and added Rule 93a to encourage use of static_assert. 8. Added Rule 110a encouraging use of the new C++11 smart pointers. 9. Removed rule 113 because it is a duplicate of rule 33. 10. Added Rule 115a encouraging	Greg Sherwin	06/03/2013

	use of C++11 nullptr 11. Added rules 116-120 encouraging use of new C++11 features.		
Error! Reference source not found.	Updated format for DMS release.	Michael Roseleip	12/09/2005
00	Initial release	Pete Landry	N/A

Table of Contents

List of Tables

1.0 Introduction

1.1 Purpose

The purpose of this document is to define *one* style of programming in C++. We are pursuing this one style of programming to achieve the following goals:

- Reduced maintenance cost
- Improve the readability and understandability of our code
- Coding consistency across our organization
- Improve the correctness and robustness of our code
- Reduce the number of "trivial" decisions programmers must make
- Make our jobs easier

Reduced Maintenance Costs

Coding standards help improve maintainability of code. It is estimated that maintenance constitutes more than 70% of the cost of any software project [10]. While coding standards alone won't abolish this number, they will certainly help reduce it.

Reading and Understanding

We write code for two audiences: computers and people. The first audience, the computer, doesn't care what style we use and will accept anything that a compiler will compile. The second audience is the driving force for these standards. Code that is easy to read, that doesn't vary in style from file to file, and is sensibly structured is easier to review, maintain, and understand.

Consistency

Every time you write a line of code you are writing for posterity, i.e. all of the programmers who will come after you. They will need to understand what you have done as well as what you intended to do. A consistent coding style for a body of code goes a long way towards helping them achieve this goal. In a study [14] that compared the effects of structure and style on expert programmers and novice programmers, it was found that inconsistencies in structure and style reduces the ability of the expert to understand the code to the same level as that of a novice. Consistency is key to our understanding of our code.

Correct and Robust Code

In addition to posterity, we also write code for ourselves. When we write code, we carry a mental model in our heads of what we *expect* the code to do. We compile the code, and then the computer shows us what our code *actually* does. Most bugs result from errors in our mental model stemming from the way we have expressed ourselves in our code. If we express ourselves in a clear, consistent style using standards that avoid the built-in obfuscations in C++, we can expect fewer bugs.

Reduce Trivial Decisions

C++ allows programmers many different options to accomplish the same result. With this freedom comes the necessity of making endless trivial choices as we write. By reducing these choices to a particular style, we can focus more on other issues, such as design and architecture.

Make Our Jobs Easier

All of the goals of this document are intended to make our jobs easier. Adhering to one consistent style of programming in our organization will make this happen.

1.2 Scope

This document applies to all Ultrasound Division software developers writing C++ code at locations as checked on the cover page. You are required, as a Ultrasound Division C++ Software Developer, to follow these rules. We hope you find them useful and enlightening.

This document will only address the C++ language. Exceptions to stated rules are described in the body of the rule.

This document has been organized into two sections: a summary of the C++ coding rules and a C++ Standards section that describes the rationale behind the rules. The summary is to be used as a checklist for coding and code inspection.

1.3 Definitions

Table 1 Definitions

Term	Definition
ANSI	American National Standards Institute
ISO	International Organization for Standardization

1.4 References

Table 2 References

Document Title	Part Number or Archive Bin Number
[1] [Clin], Cline, M., Lomow, G., *C++ FAQ*s, 1995, Addison-Wesley	N/A
[2] [Gamm], Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns,* p. 193, 1995, Addison-Wesley	N/A
[3] [Henr],Henricson, Mats, Nyquist, Erik, *Industrial Strength C++,* 1996*,* Prentice Hall	N/A
[4] [Lako], Lakos, John, *Large-Scale C++ Software Design*, Third Printing, January 1997, Addison-Wesley	N/A
[5] [Mart], Martin, Robert C., *Agile Software Development, Principles, Patterns, and Practices*, 2003, Prentice Hall	N/A
[6] [Mey1], Meyers Scott, *Effective C++*, First Edition, 1992, Addison-Wesley	N/A
[7] [Mey2], Meyers Scott, *More Effective C++*, First Edition, 1996, Addison-Wesley	N/A
[8] [Mey3], Meyers, Scott, *Effective STL*, First Edition, 2001, Addison-Wesley	N/A
[9] [McCo], McConnell, Steve, *Code Complete*, 1993, Microsoft Press	N/A
[10] [Pres], Pressman, Roger, *Software Engineerin*g, ch. 20: Software Maintenance, 1992, McGraw-Hill Inc.	N/A

[11] [Rent], *C/C++ Programming Standards*, Tim Rentsch, dated June 29, 1992	N/A
[12] [Rumb], Rumbaugh, James, *What's in a Name? A Qualified Answer*, JOOP Vol 6, No. 4, Jul/Aug 1992	N/A
[13] [Sutt], Sutter, Herb, *Exceptional C++,* Second Printing, March 2000, Addison-Wesly	N/A
[14] [Solo], Soloway, Elliot, Ehrlich, Kate, *Empirical Studies of Programming Knowledge*, 1984, TSE 10(5): 595-609	N/A
[15] [Str1], Stroustrop, Bjarne, *The Design and Evolution of C++,* 1994, Addison-Wesley	N/A
[16] [Str2], Stroustrop, Bjarne, *The C++ Programming Language*, 1997, 3rd edition, Addison-Wesley	N/A
[17] PLM Process Improvement Procedure	08658291
[18] ISO/IEC 14882:2011 Information technology – Programming languages – C++	N/A
[19] "Elements of Modern C++ Style" by Herb Sutter	Vob \BedrockDocs\Process \Guidelines&Standards\Elements of Modern C++ Style Sutter's Mill.mht (from `http://herbsutter.com/elements-of-modern-c-style/`)
[20] [Mey4], Meyers, Scott *Effective Modern C++*, First Edition, 2014, O'Reilly	N/A

1.5 Typographical Conventions

1.5.1 Syntax Conventions

C++ keywords and language constructs use a special typeface to distinguish them from the surrounding text, e.g. `class`, `new`, `operator=`.

1.5.2 Rules that result in broken or incorrect code

Rules marked with '¤' address issues with the language that will result in broken or incorrect code according to the ANSI/ISO standard.

1.5.3 C++ Standards

C++11 introduced many new language and standard library features and deprecated some existing features. Since not all compilers support C++11, rules/guidelines that presume the use of a C++11-compliant compiler and libraries are prefixed with "C++11" and where applicable an alternate rule/guideline for non-C++11 compilers is provided. Rules/guidelines that pertain only to non-C++11 compilers and libraries are prefixed with "C++98." Similarly, C++14 and C++17 specific notes will be prefixed with "C++14" and "C++17", respectively

2.0 Summary of Rules

Naming

Rule 1 Names shall be composed of complete English words or commonly recognized abbreviations and they shall reflect meaning.

Rule 2 Names consisting of more than one word are written together with each word that follows the first beginning with an uppercase letter. Special characters are not allowed.

Rule 3 Names of classes, structures, typedefs, using, and enumerated types begin with an uppercase letter.

Rule 4 Names of variables and functions begin with a lowercase letter.

Rule 4a Names of constants and enumerators begin with an uppercase letter.

Rule 5 Non-static member data shall be of the form “myParameterName”. Static member data shall be of the form “ourParameterName”. File scope data in an unnamed namespace shall be of the form “ourParameterName”.

Rule 6 Names of #defined macros and environment variables will be in all upper case and will be prefixed by package or subsystem name. Words shall be separated with underscores to improve readability.

Rule 7 An abstract base class that provides interface only must begin with the “I” prefix.

Namespace Conventions

Rule 8 Each subsystem or package shall use namespaces to isolate its names from the global namespace.

Rule 9 The use of unnamed namespaces shall be used instead of file scope statics. Unnamed namespaces shall only be used in source (e.g.,.cpp) files and not in headers.

Rule 10 The include files that correspond to the public interface of a subsystem will be stored in a dedicated subdirectory.

Rule 11 ¤ The use of *using namespace* shall be prohibited in a header file.

Files

Rule 12 All C++ header files shall have a consistent extension at a site or on a project. The use of the extension “.h” is required to facilitate code reuse.

Rule 13 All C++ source files shall have a consistent extension at a site or on a project. The use of the extension “.cpp” is required to facilitate code reuse.

Rule 14 Use the class name as the name for the header and implementation files.

Rule 15 All include files shall prevent multiple inclusion by using #pragma once.

Rule 16 Typically, classes should have a separate interface header file and implementation source file.

Rule 17 Every source file shall include its own “.h” file as the first substantive line of code.

Rule 18 Order headers in such a way to catch latent usage errors. Put in-house developed includes first. Vendor includes should go next because they are assumed to be independent. System includes are independent, so they should go last.

Rule 19	Avoid relying on one header file to include another.
Rule 20	A source file X should only include Y.h if it is required to compile.
Rule 21	In a header file, prefer forward declaration to an include directive.
Rule 22	*#include* pathnames shall not start with a "../", "./" or "/" and pathnames should not include more than one "/".
Rule 23	*#include* directives shall use quotes ("") for local (in-house) interface and header files, and angle brackets (< >) for system or vendor header files.

Comments

Rule 24 Comment style.

Error! Reference source not found.

Rule 26	Header and source code files must begin with a comment that includes a copyright and states their contents.
Rule 27	All public APIs must have a useful comment.
Rule 28	Place useful comments before complex sections of code
Rule 29	Classes or methods designed to be thread safe should include explicit comment(s) regarding thread safety.
Rule 30	Hazard mitigation code shall be commented appropriately.

Constants

Rule 31 Use `const` or `enum` instead of `#define` for constants.

Rule 31a C++11: Prefer scoped and strongly typed enums.

Error! Reference source not found..

Variables

Rule 33	Variables are to be declared with the smallest possible scope.
Rule 34	Every variable that is declared is to be given a value before it is used.
Rule 35	Use initialization instead of assignment.
Rule 36	Prefer C++ Standard basic types rather than OS API or platform specific types.

Operators

Rule 37	Prefer the pre increment operator `(++i)` to the post increment operator `(i++)`.
Rule 38 ¤	Assignment operators shall correctly handle assignment to self.

Conversions

Rule 39	C++ style casts shall be used instead of C style casts.
Rule 40	Avoid unsafe explicit casts of pointer types.
Rule 41	Do not cast away `const`.

Rule 42 Use `mutable` instead of casting away constness when member data is not part of the logical constness of a class.

Functions

Rule 43 The names of formal arguments to functions shall be specified in the function declaration and shall be the same in the function definition.

Rule 44 ¤ A function shall never return a reference or a pointer to a variable on the stack.

Rule 45 Use a `typedef` to simplify program syntax when declaring function pointers. C++ 11: Prefer `using` keyword to `typedef`.

Rule 46 A subclass shall not hide a function defined in a base class.

A subclass shall not hide a function defined in a base class.

Rule 47 Avoid the creation of overloaded virtual methods.

Const Correctness

Rule 48 Use `const&` for arguments when the function will not modify the argument.

Rule 49 Use `const wchar_t* const` instead of `wchar_t*` for string literals.

Rule 50 Class members, both functions and data members, should be declared as const whenever possible.

Overloading and Default Arguments

Rule 51 All variations of overloaded functions should be similar and used for the same purpose.

Rule 52 Use operator overloading sparingly and in a uniform manner.

Rule 53 When two operators are opposites (i.e. `==` and `!=`), define both.

Rule 54 Specify default arguments for a function in the header file only.

Inline Functions

Rule 55 Use inline functions sparingly.

Rule 56 Inline functions should be short.

Rule 57 Prefer defining inline member functions outside of the class definition.

Flow of Control

Rule 58 Do not change a loop variable inside a `for` loop block.

Rule 59 `if`, `else`, `while`, `for` and `do` must be followed by curly braces.

Rule 60 Non enumerated `switch` statements must have a `default` branch.

Rule 61 Always use inclusive lower limits and exclusive upper limits.

Rule 62 Prefer `break` to exit a loop if this avoids the use of flags.

Classes

Rule 63	public, protected, and private sections of a class must be declared explicitly and in that order.
Rule 63	public, protected, and private sections of a class must be declared explicitly and in that order.Rule 65 Avoid public member data.
Rule 66	Copy constructors and copy assignment operators should follow canonical form. C++ 11: Move constructors and move assignment operators should follow canonical form.
Rule 67 ¤	If a class should never be copied, the copy constructor and assignment operator should be declared private and left unimplemented. C++11: The unwanted member functions shall be declared public still, but also appended with `delete`. This also applies to move construction and assignment.
Rule 68 ¤	A class which dynamically allocates resources shall declare a copy constructor and assignment operator and define a destructor. C++11: If needed, this class shall also declare a move constructor and move assignment operator.
Rule 69¤	Do not overload operators new and/or delete.
Rule 70 ¤	Initialize all data members in a constructor in the order that they were declared. C++11: Use header initialization.

Inheritance

Rule 71 ¤	A base class shall define a virtual destructor.
Rule 72 ¤	An inherited default parameter value shall not be redefined.
Rule 73 ¤	Avoid calling virtual functions in a base class' constructor.
Rule 74	Do not use inheritance for "has-a" relations. **Error! Reference source not found.**.
Rule 75	Pointers or references to derived classes should always be able to be used where a pointer or reference to corresponding base class is used.
Rule 76	Use Multiple Inheritance of implementation judiciously.

Objects

Rule 77	Objects that use `new` to create other objects should be responsible for their destruction. C++ 11: Use `std::unique_ptr`, `std::shared_ptr`, or `std::weak_ptr` instead of raw pointers.
Rule 78 ¤	If a raw pointer is required use new and delete instead of malloc, calloc, realloc and free.
Rule 79 ¤	Do not delete `this`.
Rule 79 ¤	Do not delete `this` pointer to another object through the member initializer list of a constructor.
Rule 81	Avoid the creation of temporary objects.
Rule 82	Avoid interdependencies between global objects.
Rule 83	Avoid global and static data.

Error Handling

Rule 84 ¤	Prefer exceptions over fault codes whenever possible.
Rule 85 ¤	Avoid using assert in product code.
Rule 85a ¤	C++11: Prefer using static_assert.
Rule 86 ¤	Resources shall be properly released even in the presence of exceptions.
Rule 87 ¤	A function that isn't going to handle an exception shall propagate the exception to the caller.
Rule 88 ¤	Destructors shall neither throw nor allow exceptions to propagate. C++11: Mark destructors with the noexcept qualifier.

Style

Rule 89	Use consistent formatting and style within a package or subsystem.
Rule 90	Indent using four spaces.
Rule 91	Don't use tabs in code.
Rule 92	Put only one statement per line.
Rule 93	Put open and close braces on lines by themselves, indented to the same level as the code before the block (i.e., the `if`, `for`, etc. statement).

Error! Reference source not found.

Error! Reference source not found.

Rule 96	Use parentheses to clarify the order of evaluation for operators in expressions.

The C++ Standard Library

Rule 97	Prefer using the C++ Standard Library when there is a choice.
Rule 98	Prefer C++ Standard Library containers to C style arrays or customized containers.
Rule 99	Prefer Standard algorithm calls to hand-written loops when using Standard containers.
Rule 100	Prefer unicode strings to single byte strings.
Rule 101	Encapsulate raw pointers in smart pointer memory management classes.
Rule 99a	C++11: Always use the new standard smart pointers and non-owning raw pointers. Never use owning raw pointers.

Parts of C++ to Avoid

Rule 102	Avoid C-style I/O: use the `iostream` library instead.
Rule 103 ¤	Do not use `setjmp()`, `longjmp`() or goto.
Rule 104	Do not use unions of classes.
Rule 104	Do not use unions of classes.

Rule 105 **Prefer references to pointers whenever possible.**

Pointers can point to null, making them tedious with null comparisons and error-prone without. On the other hand, references cannot point to null.

This rule also covers the usage of `dynamic_cast`, which works with both pointers and references. A `dynamic_cast` to a pointer will return null if the cast fails. If we are not going to handle the invalid pointer or if we are going to handle it generically by throwing `std::bad_cast`, prefer to use a `dynamic_cast` to a reference instead. This will ensure the validity of the return by automatically throwing `std::bad_cast` if the cast has failed.

C++ 11: In general follow these guidelines for passing params that represent dynamic memory.

1) for read only access pass by `const T*` or `const T&`.

2) for transferring ownership pass `std::unique_ptr<T>` or `std::shared_ptr<T>` by value.

3) for conditionally transferring ownership pass `std::unique_ptr<T>` or `std::shared_ptr<T>` by r-value reference – should be used sparingly

Example 1 Passing dynamic resources

```
void readOnly(
        const T* p1,
        const T& p2);

void own(
        std::unique_ptr<T> uPtr,
        std::shared_ptr<T> sPtr);

void conditionallyOwn(
        std::unique_ptr<T>&& uPtr,
        std::shared_ptr<T>&& sPtr);

int main()
{
        auto uPtr = std::make_unique<T>();
        auto sPtr = std::make_shared<T>();

        readOnly(uPtr.get(), *uPtr);

        // uPtr is now moved to the param
        // and points to NULL, sPtr is copied by value and
        // ownership is now shared between main() and own().
        own(std::move(uPtr), sPtr);

        // Decisions about ownership are now delegated to
        // conditionallyOwn(). Use sparingly
        conditionallyOwn(std::move(uPtr), std::move(sPtr));

}
```

Rule 106	C++98: Pointers shall not be compared to NULL or assigned NULL; the value 0 shall be used instead.
Rule 106a	C++11: Always use nullptr for a null pointer value rather than the macro NULL or the literal 0.

C++11 Features

Rule 107	Use auto.
Rule 108	Prefer the non-member functions std::begin(), std::end(), std::cbegin(), std::cend(), std::rbegin(), std::rend(), std::crbegin(), and std::crend().
Rule 109	Prefer lambda functions whenever appropriate.
Rule 110	Support move semantics.
Rule 110	Use the uniform initialization syntax and initializer lists.
Rule 111	Use ranged for loops whenever possible.
Rule 112	Use C++ standard threading constructs whenever available.
Rule 111	Use the uniform initialization syntax and initializer lists

3.0 How to use this document

All C++ software developers should be completely familiar with all of the rules in this document. It is recommended that you begin by reading the Coding Standards Summary. If any of these rules don't make sense to you, or you disagree with them, then it is recommended that you look up the explanation of the rules that occur in the body of the document. Many of the rules are based on the books *Industrial Strength C++* by Henricson & Nyquist [3], *Effective C++* and *More Effective C++* by Meyers [6], [7], *Exceptional C++* by Sutter [13], *C++ FAQs* by Cline & Lomow [1], and *Code Complete* by McConnell [9]. Those rules that are derived from these books have a reference to the section number or page that the rule comes from if you wish further discussion of the topic. Some of the rules are marked with the symbol '¤'. These rules address issues with the language that will result in broken or incorrect code according to the ANSI/ISO standard. Many of these rules exist because our compilers do not enforce or detect them. The rest of the rules are mostly stylistic in nature. All rules are of equal importance.

4.0 Guidelines

4.1 When to break the rules

The simple answer to this topic is: *don't break the rule*s. They've been reviewed and rehashed by a lot of people and are quite functional. However, there are certain cases where breaking the rules is the only course of action:

- When you can demonstrate, *quantitativel*y, that breaking the rule results in a performance enhancement that a user would notice and appreciate as a feature.
- When breaking the rules makes the code easier to maintain by other developers. If you find that you are breaking a rule all of the time, and you feel that you have a very good reason for breaking it, then lobby to change the rule. Your voice counts. We will pay attention to you. See the next section for details on how to do this.

If you choose to break a rule, comment the code to document why the rule was broken.

4.2 How to change the rules

This is a living document. As the quality of our compilers improves to catch up to the ANSI/ISO standard, certain rules will become obsolete or unnecessary. In addition, coding habits evolve. What may seem like gospel this year may prove to be different as our organization matures in its use of C++.

Process improvements are to be submitted via the PLM Process Improvement Procedure [17]. Your proposal will be put to review and incorporated if it is worthwhile. It behooves you to research your proposal first. Backing up your ideas with studies or notable books goes a long way to a smooth acceptance.

5.0 Detailed Rule Descriptions

5.1 Naming

Naming is the most important aspect of C++ programming style. It is through carefully chosen names that we convey the primary intent of our code. The names chosen for the variables, functions, files, classes, etc. in our system are the first place programmers look to for clarity. The rules here are to help make this clarity a reality.

Rule 1 Names shall be composed of complete English words or commonly recognized abbreviations and they shall reflect meaning.

Of all the naming rules in this document, this one is probably the most important. One heuristic is that a name, which cannot be pronounced, is a bad name. A long name is preferred over a short, cryptic name. Arbitrarily dropping vowels from a name is not allowed. Avoid abbreviations unless they are part of the English language or are a standard part of the lexicon of computer science, i.e. 'id' instead of 'identification' or the indexing variables i, j, and k. Studies have shown that abbreviation leads to less maintainable code. When in doubt, spell it out. Abbreviating for the sake of saving typing time is indefensible, since the amount of time actually spent typing in a name is insignificant compared to time wasted decoding unreadable code. Global variables, functions, and constants ought to have long enough names that their purpose is self-explanatory.

Example 2 Examples of names

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

Example 3 Ambiguous names

```
void termProcess(); // Terminate process or
                    // terminal process?
```

Example 4 Names having numeric characters can be difficult to read.

```
int i0 = 13; // Names with digits can be
int i0 = iO; // difficult to read.(Izero vs.IO)
```

[Henr:1.1; Rent:pg. 8]

Rule 2 Names consisting of more than one word are written together with each word that follows the first beginning with an uppercase letter. Special characters are not allowed.

This rule helps in clustering identifiers visually. It also has the advantage that it does not require special characters so names can be used across a wide range of languages.

An exception to this rule applies when doing database programming. In that case, it is appropriate to use the naming conventions of the database when the variable corresponds directly to a table or column in the database.

[Henr:A.2; Rent:pg. 9]

Rule 3 **Names of classes, structures, typedefs, using, and enumerated types begin with an uppercase letter.**

[Henr:A.3; Rent:pg. 9]

Rule 4 **Names of variables and functions begin with a lowercase letter.**

[Henr:A.4]

Rule 4a **Names of constants and enumerators begin with an uppercase letter.**

Rule 5 **Non-static member data shall be of the form “myParameterName”. Static member data shall be of the form “ourParameterName”. File scope data in an unnamed namespace shall be of the form “ourParameterName”.**

This makes code more readable by making it clear which symbols in a method body are members or file scope data and which are not. It eliminates problems with similar names in constructors as well as function/member data name conflicts. This rule does not apply to public class constants.

Rule 6 **Names of #defined macros and environment variables will be in all upper case and will be prefixed by package or subsystem name. Words shall be separated with underscores to improve readability.**

This provides a simple means of identifying macros. In general, macros should be avoided, but when they do need to be created, name them all uppercase.

Example 5 #define naming convention

```
#define CLIP_RECORDING_CONTROLLER_H_ALREADY_INCLUDED
```

[Henr:A.6; Rent:pg. 9]

Rule 7 **An abstract base class that provides interface only must begin with the “I” prefix.**

This makes code easier to understand by having a consistent approach that conveys more of the purpose of a base class.

An interface class only includes pure virtual functions and does not include data members.

Example 6 Interface and abstract base class naming

```
IClipPlaybackControl
```

5.2 Namespace Conventions

Rule 8 Each subsystem or package shall use namespaces to isolate its names from the global namespace.

Using the namespace feature will help avoid name clashes in the global namespace. The namespace feature also promotes modularization and is a good tool for identifying subsystem/module interfaces..

Rule 9 The use of unnamed namespaces shall be used instead of file scope statics. Unnamed namespaces shall only be used in source (e.g.,.cpp) files and not in headers.

Rule 10 The include files that correspond to the public interface of a subsystem will be stored in a dedicated subdirectory.

Rule 11 ¤ The use of *using namespace* shall be prohibited in a header file.

When *using namespace* is placed in a header file, it makes it difficult to control the scope of the *using namespace* directive and can cause ambiguity for symbols.

5.3 Files

The physical rules for creating classes in files are not only important to the developer when navigating around the code, but also to the development environment, which makes assumptions about naming conventions to support compilers, make, etc.

Rule 12 All C++ header files shall have a consistent extension at a site or on a project. The use of the extension ".h" is required to facilitate code reuse.

The purpose of this convention is to provide a uniform interpretation of header file names. Any file containing hazard mitigation code shall include _Mitigation in the name: UltrasoundFunction_Mitigation.h

[Henr: A.9]

Rule 13 All C++ source files shall have a consistent extension at a site or on a project. The use of the extension ".cpp" is required to facilitate code reuse.

The purpose of this convention is to provide a uniform interpretation of source file names. Any file containing hazard mitigation code shall include _Mitigation in the name: UltrasoundFunction_Mitigation.cpp

[Henr: A.10]

Rule 14 Use the class name as the name for the header and implementation files.

Using the class name as the file name makes the class header and implementation easier to find. Implementation files that must be split for a specific purpose should have a filename suffix that shows the intent of that purpose (e.g., a file put in a directory that contains implementation that has external linkage could have an "External suffix").

Rule 15 **All include files shall prevent multiple inclusion by using #pragma once.**

Example 7 A compiler may read a header file multiple times for a single source file because of nested includes. Multiple declarations of a class are errors (otherwise you could use the same name for different classes).Header with include guards

```
#pragma once

//
// Copyright (c) 2016-2016 by Siemens Medical Solutions, Inc.
// All Rights Reserved.
//
// No part of this software may be reproduced or transmitted in
// any form or by any means including photocopying or recording
// without written permission of the copyright owner.
//
```

[Henr:2.3; Rent:pg. 31]

Rule 16 **Typically, classes should have a separate interface header file and implementation source file.**

Separation of class interface and implementation allow the implementation to remain hidden from callers. Exceptions that do not require separate header and source files or that can be implemented in the header or source file of another class include function objects, classes that implement the pImpl part of the pImpl idiom [Sutt: Item 29], templatized classes requiring the implementation to be in-line for it to compile, and small classes where implementation is included so that linkages between libraries can be minimized.

Rule 17 **Every source file shall include its own ".h" file as the first substantive line of code.**

This rule will help avoid latent usage errors. If the ".h" file is not included first in its own implementation file, then it may not be complete (i.e., it may be using symbols or definitions defined before it in the implementation file). Putting the ".h" first ensures the completeness of the interface (at least from a compilation perspective). The exception to this rule is that precompiled headers must always be the first included header. In this case, the header of the same name shall be the second include.

[Lako: pp. 110-112]

Rule 18 **Order headers in such a way to catch latent usage errors. Put in-house developed includes first. Vendor includes should go next because they are assumed to be independent. System includes are independent, so they should go last.**

The goal is to prevent #include order dependencies within in-house developed code. Include directives should never be order dependent, because the code would be very difficult to maintain. By ordering #include directives in a way that those that are more likely to be dependent on other headers or symbols are first, we increase the probability of catching incomplete header files with dependencies on other headers.

Rule 19 **Avoid relying on one header file to include another.**

By not relying on one header to include another, the code is more maintainable. For example, there might be a class header, Part.h, that #includes <string>. An engineer creates an Assembly class and #includes Part.h in its header. The Assembly class also has string member data. If the Assembly class does not #include <string> in its header then it is relying on the fact that Part.h includes <string>. In this way it is becoming dependent on the implementation of Part. If Part is changed so that string is no longer included then Assembly will no longer compile. If <string> had been included directly in the Assembly class header, the problem would not have occurred.

The exception is in the use of a derived class. It is okay for a client to assume that a derived class will contain all that is required to use the base class part of the class because the derived header is a logical extension of the base class interface.

[Lako: pp. 113-114]

Rule 20 **A source file X should only include Y.h if it is required to compile.**

Unnecessary includes in source files needlessly increase compile times.

[Lako:pp. 135-136]

Rule 21 **In a header file, prefer forward declaration to an include directive.**

In a number of cases (e.g., classes that are only accessed via pointers or references), a forward declaration is all that is required to allow a header to compile. Forward declarations do not add physical file dependencies to the system, whereas #include directives do.

[Henr: 2.2, Mey1: Item 34, Sutt: Item 26]

Rule 22 ***#include* pathnames shall not start with a “../”, “./” or “/” and pathnames should not include more than one “/”.**

All of these methods of inclusion will break if the file doing the including, or the file being included is moved in the file system.

[Henr: 15.5]

Rule 23 ***#include* directives shall use quotes (“”) for local (in-house) interface and header files, and angle brackets (< >) for system or vendor header files.**

This convention will allow us to determine at a glance whether an include file is a application include file or a system include file.

Note that the -I preprocessor option applies to both ““and <> included headers.

Example 8 Use of brackets and quotes in include file names

```
#include "MyFile.h"
#include <string>
```

[Henr: 15.4]

5.4 Comments

Comments are your chance to say something that goes beyond the formal description of what the code is telling the machine. Good comments don’t repeat the code. They clarify its intent. They explain the code at a higher abstract level than the programming language provides. Some of the rules in this section deal with the mechanics of writing comments, others with their style. Pay attention to all of them. For an excellent discussion on how and why to comment, see [9], ch. 19. It is strongly recommended that you read this chapter.

Rule 24 Comment style

One line comments should be placed above the code they are commenting and indented to the same level. Comments to the right of each line of code are discouraged, as there is usually insufficient space for a meaningful comment and the code begins to look cluttered. Vertical whitespace before and after comments is OK. Block comments should also be placed above the code they are commenting and indented to the same level.

Example 9 Inline comment style

```
// Useful one line comment describing next few lines
foo = function(slope, offset);
bar = intercept(foo, x, y);
```

Example 10 Block comment style

```
// The following loop does something worth noting to
// another programmer that may relate to a higher,
// abstract concept that the loop doesn't quite express
while (function(foo, bar))
{
        ...;
}
```

Rule 25 Use // for comments.

An exception to this rule is if your header file will be included by a C program, at which point you must use C-style comments.

[Henr: 3.4, Mey1: Item 4]

Rule 26 Header and source code files must begin with a comment that includes a copyright and states their contents.

All our code must have a copyright statement to protect the legal rights of Siemens. If the code has been developed over a period of years, each year must be stated in the copyright notice. The author must include a comment block within the files they create as shown in the following example.

Example 11 Required copyright - source or header file

```
//
// Copyright (c) 2016-2016 by Siemens Medical Solutions, Inc.
// All Rights Reserved.
//
// No part of this software may be reproduced or transmitted in
// any form or by any means including photocopying or recording
// without written permission of the copyright owner.
//
```

[Henr: 3.1, 3.2]

Rule 27 All public APIs must have a useful comment.

The comment should contain a description of the function, what its parameters mean, and what the function returns. Exceptions not requiring comments are trivial functions such as accessors.

Rule 28 Place useful comments before complex sections of code

Block comments before a particularly difficult or obscure piece of code or a complicated loop are encouraged. Detailed comments before constructs such as in-line assembly are a must.

Rule 29 Classes or methods designed to be thread safe should include explicit comment(s) regarding thread safety.

Since many classes will be required to work properly in a multithreaded environment it is important to know which classes or methods were created for use in multithreading. For classes where multithreading is not within scope, comments on thread safety (e.g., Designed for Single Threaded Usage or NOT Thread Safe) are optional.

Rule 30 Hazard mitigation code shall be commented appropriately.

Example 12 A C++ file header where the entire file implements a risk control measure requirement

```
//
// Copyright (c) 2017-2017 by Siemens Medical Solutions, Inc.
// All Rights Reserved.
//
// No part of this software may be reproduced or transmitted in
// any form or by any means including photocopying or recording
// without written permission of the copyright owner.
//
// Contents: <contents description>
//
// Mitigation: This class implements or tests a risk control measure.
//             Do not modify without formal code review.
// tracetag: <requirement ID or trace tag>
```

Example 13 A C++ method header where a single method is related to the implementation of a risk control measure requirement

```
//<Normal method header description>
// Mitigation: This method implements or tests a risk control measure. Do
// not modify without formal code review.
// tracetag: <requirement ID or trace tag>
```

5.5 Constants

Constants came from the desire for a read-only data mechanism for compilers [Str1: Section 3.8]. It has developed into a means for the programmer to strengthen the already strong type-checking of C++. The rules in this section apply to constants in the code. Const-correct classes and functions are discussed in the section on const correctness.

Rule 31 Use `const` or `enum` instead of `#define` for constants.

The C++ model is strongly typed. Using #define goes around this, which makes it harder for the compiler to detect errors. In addition, consts are visible in debuggers and #define values are typically not.

Example 14 Different ways of declaring constants.

```
// Constants using macros
#define BUFFERSIZE 7 // Wrong. No type checking
```

```
// Constants using const
const std::size_t BufferSize = 7; // Right.
```

[Henr: 13.5; Mey1: Items 1, 21]

Rule 31a **C++11: Prefer scoped and strongly typed enums.**

Enum classes (i.e. strongly named enums) address three problems with traditional C++ enum – implicit conversion to int; enumerator names are exported to the surrounding scope; and the underlying type of the enumerators cannot be specified.

Example 15 **Strongly typed enums.**

```
enum Alert { green, yellow, election, red }; // traditional enum
enum class Color { red, blue }; // scoped and strongly typed enum
                                // no export of enumerator names into
                                // enclosing scope.
                                // no implicit conversion to int
Alert a = 7; // error
Color c = 7; // error: no int->Color conversions
int a2 = red; // OK: Alert->int conversions
int a3 = Alert::red; // erron in C++98; OK in C++11
int a4 = blue; // erron: blue not in scope
int a5 = Color::blue; // error: no Color->int conversion
Color a6 = Color::blue; // OK
enum class TrafficLight : char { red, yellow, green }; // type is char
enum class Color_code : char // forward declaration
void foo(Color_code* cc); // use of forward declaration
```

Rule 32 **Symbolic constant values shall be used instead of literal constants.**

Using symbolic values through the const or enum mechanism allows the numeric constants in the code to be changed in one place without requiring an onerous search for all occurrences of a fixed numeric value. Symbolic values also make the code clearer to understand, since the limit they are imposing can be clearly delimited in the name of the value.

There are situations, however, where the use of literal constants is preferred over symbolic values. These include initializing an array index to zero, Rule 115 (pointers and zero), and other cases where numeric literals have a well-established and clear meaning.

Example 16 **Correct/incorrect usage of literals**

```
for (int i = 0; i < TableSize; ++i) // use of 0 is OK
for (int i = 0; i < 452; ++i) // undesirable

int
average(int x, int y)
{
       return (x + y) / 2; // OK, no reasonable alternative
}

if (bufferP == 0) // C++98 OK, pointer should not be compared to NULL
if (!bufferP) // Preferred, no need to compare to 0 or NULL
if (bufferP != nullptr) // C++ 11

foo(bufferP, ControlNormal); // correct
foo(bufferP, 2); // undesirable, cryptic
```

[Henr: 5.1]

5.6 Variables

These rules should help keep your code correct as well as improve its performance.

Rule 33 Variables are to be declared with the smallest possible scope.

This prevents variables from being unnecessarily allocated as well as improves the readability of the code. It is also recommended that variables be declared close to where they are used.

[Henr: 5.1; Mey1: Item 32]

Rule 34 Every variable that is declared is to be given a value before it is used.

A variable must always be initialized before use. The compiler will normally give a warning if a variable is used before it is assigned, so this recommendation should be easy to follow.

Rule 35 Use initialization instead of assignment.

By always initializing objects 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. Even for intrinsic types this is a good habit to get into. Using initialization in constructors is particularly important for reasons of efficiency.

Example 17 Using initialization instead of assignment

```
int i; // Wrong.

//... 1022 lines of code
i = 10;

int j = 10; // Right.

// C++ 11 and later:
int j { 10 }; // Braced initialization equivalent to above
int k[100] {}; // Equivalent to 0 filling

class MyClass
{
public:
       MyClass(const wchar_t* initialString);
private:
       std::wstring myPrivateString;
       int myComplicated;
};

MyClass::MyClass(const wchar_t* initialString) // Wrong.
{
       myPrivateString = initialString;
       ...
}
MyClass::MyClass(const wchar_t* initialString) : // Right
       myPrivateString(initialString) // Right
{
}
```

Rule 36 Prefer C++ Standard basic types rather than OS API or platform specific types.

Example 18 Prefer C++ Standard types over OS API or platform specific types

Use bool (C++ specific) instead of BOOL (MFC specific or your own home-brewed version).

Use std::wstring instead of CString.

Use std::int64_t instead of __int64.

5.7 Operators

Rule 37 Prefer the pre increment operator `(++i)` to the post increment operator `(i++)`.

The expression i++ returns a copy of the original state of i, which requires that (in most cases) an unnecessary copy of the old value of i be created, which means that the destructor for that copy will also be called. Some compilers will optimize this behavior for built-in types, but this will not happen for user defined types. This can induce considerable overhead for such a simple operator. Thus, using the pre-increment is more efficient and should be used unless the specific behavior of the post-increment is needed.

[Clin: Question 371; Mey2: Item 6]

Rule 38 ¤ Assignment operators shall correctly handle assignment to self.

During the assignment of an object, a = b, the assignment operator typically must first perform the destructor operation on the left hand object and then copy the attributes from the right hand object. If, however, the assignment is a = a then the destructor operation must not occur because there would not be anything left to assign. One can check for assignment to self by doing something similar to:

Example 19 Checking for assignment to self

```
X&
X::operator=(const X& rhs)
{
	if (*this == rhs)
	{
		return *this;
	}
	...
}
```

In the case where the assignment operator provides a strong exception safety guarantee (commit/rollback semantics) the check for self-assignment is not required and would be for the purpose of optimization only.

Example 20 Ideally, a good copy constructor would make use of the copy-and-swap idiom. The copy-and-swap idiom allows for strong exception guarantees and allows us to avoid checking for self-assignment.Copy-and-swap idiom

```
class CopyAndSwap

{
public:
	// Define copy constructor as usual
	CopyAndSwap(const CopyAndSwap& other) { ... }

	CopyAndSwap& operator=(CopyAndSwap other)
	{
```

```
                swap(*this, other);

                return *this;
        }

        friend void swap(CopyAndSwap& first, CopyAndSwap& second)
        {
                using std::swap;

                swap(first.member1, second.member1);
                swap(first.member2, second.member2);
        }
};
```

In the copy-and-swap idiom, we define a member swap function that will take care of swapping all of the members between two instances.

The copy assignment operator will call the member swap function with the current instance of the class as well as the instance that is passed in.

The copy assignment operator will also take the parameter by value instead of by reference. This allows the compiler to construct a temporary object by delegating the work to the already defined copy constructor, swap the members of the temporary's instance with our current instance, and finally destroy the temporary instance once the end of the copy assignment scope has been reached.

This allows for strong exception guarantees in the case of failures (e.g. failed allocations) and gives us an elegant way of delegating copy construction work without checking for self-assignment.

C++ 11: A similar design could be employed for move constructors with a move-and-swap idiom. However, this is not as effective since the move assignment operator would require the construction of a temporary object which could avoided with the use of rvalue references.

[Henr: 7.7; Mey1:Items 15, 17; Sutt Item 38].

5.8 Conversions

In general, casting away the type of anything is a *really bad ide*a, and should be avoided at all costs.

Rule 39 C++ style casts shall be used instead of C style casts.

The C++ style casts (static_cast, dynamic_cast, reinterpret_cast, const_cast) are easier to parse (both for humans and tools), and they allow compilers to diagnose casting errors that would otherwise go undetected.

[Mey2: Item 2]

Example 21 C++ style casts:

```
int x { 5 };
int y { 2 };

// Wrong
float quotient = (float) x / (float) y;

// Right
float quotient =
    static_cast< float >(x) / static_cast< float >(y);
```

Rule 40 Avoid unsafe explicit casts of pointer types.

Casting should be avoided purely because such code can cause memory errors. The compiler will generally do the right thing when asked to convert an integral or floating point type to some other integral or floating point type (i.e., float to long). The compiler will complain about being asked to implicitly convert a non-`void` pointer type to any other non-`void` pointer type. The reason that the compiler complains is a good one: converting between arbitrary pointer types is very dangerous. Converting an `int*` to a `double*` will certainly not provide the desired results.

[Henr: 5.1, Mey1: Item 39]

Rule 41 Do not cast away `const`.

There is no reason to cast away const-ness other than to accommodate a poorly designed 3rd party or vendor supplied function that lacks proper const arguments. Casting away const-ness breaks the rules associated with the variable or function declared const and allows the caster to corrupt data without the compiler catching the error.

[Henr: 6.3, Mey1: Item 21]

Rule 42 Use `mutable` instead of casting away constness when member data is not part of the logical constness of a class.

The keyword mutable was created for those few instances where some part of the state of an object is logically not part of the constness of that object. For example, consider a Buffer class that has a mutex data member used to protect the buffer from simultaneous write operations in multiple threads. The mutex in no way describes the logical state of the buffer (size, location, etc.). Rather, it provides a locking mechanism to protect the logical state. In this case, the mutex would be declared `mutable` in the class header and const methods of the Buffer would be able to operate on the Buffer instance and change the locking state as appropriate.

5.9 Functions

The way we declare and use functions in our code is key to its readability and functionality. The rules in this section apply to stand-alone functions as well as member functions and should help improve the clarity and robustness of your code.

Rule 43 The names of formal arguments to functions shall be specified in the function declaration and shall be the same in the function definition.

This is for consistency and ease of understanding.

If an argument is not used in the function body then an exception is allowed. In this case the argument name may appear in the declaration but shall not appear in the definition to avoid compiler warnings.

Example 22 Declaration of formal arguments

```
int setPoint(int, int); // Wrong.
int setPoint(int x, int y); // Right.

int
setPoint(int x, int y)
{
        // ...
}
```

[Henr: A.11]

Rule 44 ¤ A function shall never return a reference or a pointer to a variable on the stack.

Local variables are allocated storage when a function is invoked (in scope) and storage is deallocated when the function is exited (goes out of scope).When the function goes out of scope the memory location used by the local variable may be used by some other entity and therefore the contents of the addressed memory are unpredictable.

Example 23 Incorrect return of a variable on the stack

```
wchar_t* trouble()
{
        wchar_t localString[24];
        // do something
        return localString; // bad news
}
```

[Henr: 5.9, Mey1: Item 31]

Rule 45 Use a `typedef` to simplify program syntax when declaring function pointers. C++ 11: Prefer `using` keyword to `typedef`.

In general, inheritance, polymorphism, and virtual methods make function pointers unnecessary, but occasionally a function pointer is needed. If you are using function pointers consider whether or not you should be using dynamic binding instead. When you do have to use function pointers, using `typedef` is a good way of making code more easily maintainable and portable.

Example 24 Syntax simplification of function pointers using a typedef

```
// ‘Ordinary’ way of declaring pointers to functions:
double (*mathFunction)(double) = sqrt; // Wrong.

// With a typedef, things are much more readable.
typedef double MathFunctionType(double); // Right.

MathFunctionType* mathFunction = sqrt;

void main()
{
        // You invoke the function like this:
        double returnValue = mathFunction(23.0);
}

// C++ 11 and later:
using MathFunctionType = double(double);
```

Rule 46 A subclass shall not hide a function defined in a base class.

Virtual functions are dynamically bound, while non-virtual functions are statically bound. When you redefine a non-virtual function in a subclass you restrict yourself by relying on the object pointer type (static type) to determine which function (the one in the base class vs. the one in the subclass) is executed when called.

[Mey1: Item 37]

Rule 47 **Avoid the creation of overloaded virtual methods.**

This rule will help to ensure intuitive and proper polymorphic behavior of a derived class. When a derived class overrides some, but not all, of a number of overloaded methods in the base class, method name hiding occurs. This name hiding is usually not intended nor is it desirable as it can lead to unexpected behavior.

5.10 Const Correctness

Const correctness in a program can go a long way to insuring that the program is doing what you expect it to do. Const is such a valuable feature of C++'s strong typing that it should be used wherever possible [6].

Rule 48 **Use `const&` for arguments when the function will not modify the argument.**

[Henr: 7.8, Mey1: Item 21]

Rule 49 **Use `const wchar_t* const` instead of `wchar_t*` for string literals.**

If you have to declare a string literal in the program, use const wchar_t*. This accurately reflects the use of the literal in your code and allows the compiler to intercept possible memory corruption errors before they occur. Note this rule applies equally when char* is used.

Example 25 **Use of const with string literals**

```
wchar_t* baseName = L"Rickenbacker"; // Wrong.
const wchar_t* const baseName = L"Fender"; // Right.

// C++ 11: Use if possible
constexpr wchar_t* message = { L"Message" };
```

[Henr: 7.10]

Rule 50 **Class members, both functions and data members, should be declared as const whenever possible.**

This rule produces programs that are easier to comprehend and allows more thorough compile-time checking of program invariants.

Member functions declared as *const* may not modify member data and are the only functions that can be invoked on *const* objects. An exception is that *const* member functions can modify member data that is declared mutable because mutable members are not considered as part of the constness of the class.

An example of a typical violation of this rule is to have a method declared const and to access a singleton object in the method body which changes system state. Note that C++ allows a const member function to be overloaded with a non-const member function.

Example 26 **const-declared access functions to internal data in a class**

```
class SpecialAccount : public Account
{
public:
       void insertMoney(int moneyAmount);

       int getAmountOfMoney(); // Wrong.
```

```
        int getAmountOfMoney() const; // Right.

private:
        int myMoneyAmount;
};
```

Example 27 Overloading an operator/function with respect to const-ness

```
class Buffer
{
public:
        Buffer(const char* initialString);
        ~Buffer();

        char& operator[](unsigned index); // returns an lvalue.
        char operator[](unsigned index) const; // returns an rvalue.

private:
        char* myBuffer;
};

int main()
{
        const Buffer constName("peter"); // This is a
                                          // constant buffer
        Buffer name("mary"); // This buffer can change

        name[2] = 'c'; // This works.
        constName[2] = 'c' // Correctly generates a compiler error.
}
```

[Henr: 7.11, 7.13; Mey1:Item 21; Rent:pg. 16]

5.11 Overloading and Default Arguments

Rule 51 All variations of overloaded functions should be similar and used for the same purpose.

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 cause considerable confusion.

Example 28 Example of the proper usage of function overloading

```
class String
{
public: // Used like this:
        // ... // String x = "abc123";
        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(x);
};
```

[Henr: 7.14]

Rule 52 Use operator overloading sparingly and in a uniform manner.

Operator overloading has both advantages and disadvantages. While operator overloading may permit more compact and legible code, be careful to avoid using operator overloading with a class where it does not make sense semantically. If the semantics are simple and natural, code written using the overloaded operators will generally be clear and easy to understand. An example of appropriate operator usage and overloading would be a complex number class. In this case, the semantic meanings of the operators make sense within the context of the class, so it is appropriate to use operator overloading. When defining the operators it is important to define them all, e.g. don't define '+' and not define '-', '*', '/', etc.

[Henr: 7.15]

Rule 53 When two operators are opposites (i.e. == and !=), define both.

If the operator != has been designed for a class, then a user may well be surprised if the operator == is not defined as well. It is recommended that the second operator be defined using the first.

Rule 54 Specify default arguments for a function in the header file only.

This is in accordance with the ANSI/ISO C++ Standard.

[Henr: 7.17]

5.12 Inline Functions

Rule 55 Use inline functions sparingly.

Inlining is not guaranteed. Your compiler may decide that instead of inlining the function it will create your inline function in every translation unit (.o or .obj file) that includes the header file. This can cause code bloat. Defining functions in a class declaration exposes the details of the implementation. This is acceptable for simple accessor functions, but should generally be avoided. Your compiler may have an option that will give you a warning if it is not properly inlining a function. Furthermore, when inlining you should quantitatively evaluate the performance benefit of the inline function. In most cases, the performance benefit has little or no bearing on the overall performance of the code.

[Mey1: Item 33]

Rule 56 Inline functions should be short.

Larger functions may encourage your compiler to not inline. Any function that is longer than two statements is probably too complex to derive any serious performance benefit. More complex functions are also more likely to be modified as the code grows.

Rule 57 Prefer defining inline member functions outside of the class definition.

When making a decision where to put an in-lined method definition consider the readability of the header. Generally, it is considered good practice to place all inline member functions at the end of the header file. They can clutter up the class definition making the interface harder to read. Also, by defining inlined methods outside of the class definition, the inlinedness of the methods can be revoked or reasserted easily without disturbing the core declaration of the class. A possible exception is a trivial one-line accessor method.

[Henr: A.15]

5.13 Flow of Control

Rule 58 **Do not change a loop variable inside a `for` loop block.**

It is very confusing and error-prone to change the loop variable in a for loop inside the loop body.

[Henr: 4.1]

Rule 59 `if`, `else`, `while`, `for` **and** `do` **must be followed by curly braces.**

This applies to both single statements and ';' statements. Single statement blocks should always be surrounded by curly braces. Ignoring this rule can lead to subtle flow of control errors that can be avoided by the rigorous use of curly braces. Experience has shown that this is a recurring bug in coding shops that allow this practice.

Example 29 **Flow control structure without statements**

```
if (myTest) // Wrong.
        doSomething();

if (myTest) // Right.
{
        doSomething();
}

while (something); // Wrong.
while (something) // Right.
{
        // Empty block
}
```

[Henr: 4.3]

Rule 60 **Non enumerated `switch` statements must have a `default` branch.**

As code grows, it is not uncommon for the need to add new cases to switch statements. A lot of time this is overlooked by the hurried programmer. Always including a default branch will catch these conditions. For those switch statements where the switch is performed on an enumerated type, it is preferable to *not* have a default clause.

[Henr: 4.5]

Rule 61 **Always use inclusive lower limits and exclusive upper limits.**

Since C++ arrays start their numbering at 0, it is best to use inclusive lower and exclusive upper limits. Instead of saying that `x` is in the interval `x >= 23` and `x <= 42`, use the limits `x >= 23` and `x < 43`. The following important claims then apply:

- The size of the interval between the limits is the difference between the limits.
- The limits are equal if the interval is empty.
- The upper limit is never less than the lower limit.

By being consistent in this regard, many difficult errors will be avoided.

Example 30 Limits for loop variables

```
int array[10];
int ten = 10;
int nine = 9;

for (int i = 0; i < ten; ++i) // Right.
{
        array[i] = 0;
}

for (int j = 0; j <= nine; ++j) // Wrong.
{
        array[j] = 0;
}
```

Rule 62 Prefer `break` to exit a loop if this avoids the use of flags.

This can help make code more readable and eliminate some ugly conditionals. Remember that break breaks you out of the innermost loop containing the break.

Example 31 Using break to exit a loop, no flags are needed.

```
do // Right.
{
        if (something)
        {
                // Do something
                break;
        }
} while (someCondition);

bool done = false; // Wrong.
do
{
        if (something)
        {
                // Do something
                done = true;
        }
} while (someCondition && !done);
```

5.14 Classes

Rule 63 `public`, `protected`, and `private` sections of a class must be declared explicitly and in that order.

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. All elements in a class should have explicit member access control. The member access control labels should be indented two (2) spaces and the class members should be indented four (4) spaces beyond the opening brace. The names of the member data and member functions can be aligned if desired.

Example 32 Use explicit member access controls in function declarations

```
class Widget
{
public:
       Widget();
       Widget(const Widget& other);
       ~Widget();

       unsigned int getLength() const;
       const wchar_t* getString() const;
       ...

protected:
       ...

private:
       unsigned int myLength;
       wchar_t* myString;
};
```

[Henr: A.13]

Rule 64 **The order of members shall be all default constructor, default destructor, all other constructors (including copy constructor, move constructor), copy assignment operator, move assignment operator, all other overloaded operators, then member functions. The order of member variables shall be in order of increasing object size.**

Example 33 Order of members

```
class CorrectOrder
{
        CorrectOrder();
        ~CorrectOrder();

        CorrectOrder(int element) :
                mySmallElement(element) { ... }

        CorrectOrder(const CorrectOrder& other);
        CorrectOrder(CorrectOrder&& other);

        CorrectOrder& operator=(const CorrectOrder& rhs);
        CorrectOrder& operator=(CorrectOrder&& rhs);

        // All other operators
        bool operator==(...) const;

        // All other functions
        void doSomething();

private:

        // Order of member variables from smaller to bigger objects
        int mySmallElement = 0;
        double mySlightlyBiggerElement = 0.0;

        std::wstring myMediumSizedObject { L"MediumSizedObject" };
        ReallyBigObject myBigObject;
};
```

Rule 65 **Avoid public member data.**

The hiding of implementation is one of the most important benefits of object-oriented design and programming. The actual data representation of a class is an implementation detail that must not be public.

If access to member data is required outside a class then member functions should be defined to provide that access. Access functions should only be provided for those member data that are intrinsically part of the problem domain, never part of the solution domain. However, none of the above should be taken to mean that the use of such data access functions is encouraged, only that it is preferable to direct access. The presence of many “get” and “set” functions should be questioned and, perhaps, the design should be rethought.

This reasoning also applies to protected data members. However, since the scope of the protected data members is restricted to subclasses, their use is allowed, although not recommended.

[Mey1: Item 20; Rent: pg. 27; Henr: 10.1]

Rule 66 **Copy constructors and copy assignment operators should follow canonical form. C++ 11: Move constructors and move assignment operators should follow canonical form.**

A canonical assignment operator in class X should have the signature:

Example 34 **Canonical assignment operator**

```
X& X::operator=(const X& rhs);
```

To be consistent with both the compiler’s behavior for built-in types and the behavior of the assignment operators in the standard library, it is necessary for assignment operators to return non-const values.

The assignment operator may also be overloaded to take different types of arguments but it should always return a non-const reference to *this.

A canonical copy constructor in class X should have the signature:

Example 35 **Canonical copy constructor**

```
X::X(const X& other);
```

A canonical move assignment operator in class X should have the signature:

Example 36 **Canonical move assignment operator**

```
X& X::operator=(X&& rhs);
```

A canonical move constructor in class X should have the signature:

Example 37 **Canonical move constructor**

```
X::X(X&& other);
```

Example 38 **Canonical copy and move assignment operator if using the swap idiom**

```
X& X::operator=(X rhs);
```

This assignment operator will work for both copy and move semantics. The construction by value can happen either through a copy or a move.

In the case of move, an additional construction will take place. This construction could have been avoided with the use of rvalue references.

[Henr: 7.9]

Rule 67 ¤ **If a class should never be copied, the copy constructor and assignment operator should be declared private and left unimplemented. C++11: The unwanted member functions shall be declared public still, but also appended with `delete`. This also applies to move construction and assignment.**

If you don't do this, the compiler will generate these for you. By using this technique, you avoid any accidental calls to these routines - the compiler will flag them as errors.

[Henr: 5.10, Mey1: Item 27]

Example 39 Deleted move constructor

```
X::X(X&&) = delete;
```

Rule 68 ¤ **A class which dynamically allocates resources shall declare a copy constructor and assignment operator and define a destructor. C++11: If needed, this class shall also declare a move constructor and move assignment operator**

By "resource" we mean anything that is dynamically allocated and can, and should, be deallocated when no longer needed. The prototypical example is dynamically allocated memory but it may also include OS system resources, etc.

The compiler automatically provides a copy constructor if one is not provided by the class (even if you provide other constructors). The built-in copy constructor provided by the compiler uses what is called *memberwise initialization*. Memberwise initialization recursively applies the copy constructors of the class data members but it performs a bitwise copy of the primitive types (char, int, pointers, etc.). The bitwise copying of the primitive types can wreck havoc if those values represent dynamically allocated resources which are subsequently deallocated by the destructor: the destructor may attempt to deallocate the same resource twice! A user-defined copy constructor is needed to avoid this situation.

The compiler automatically provides an assignment operator if one is not provided by the class. The built-in assignment operator provided by the compiler uses ***memberwise assignment***. Memberwise assignment recursively applies the assignment operator of the class data members but it performs a bitwise copy of the primitive types. Memberwise assignment suffers from the same potential problem as memberwise initialization discussed above. Again, a user-defined assignment operator is needed.

It is up to the class designer to decide if it is best to share resources (shallow copy) when objects are copied or assigned or whether the resources themselves should be copied (deep copy).

If you do not wish to provide copy and assignment operators for a class then they should be declared private and no implementation should be given.

Example 40 Definition of a "dangerous" class not having a copy constructor

```
// "Dangerous" Widget class
class Widget
{
public:
	Widget(const wchar_t* initialString);
	~Widget();

	// Wrong. No copy constructor.
	// Wrong. No operator=
private:
	wchar_t* myValue;
};
```

```
Widget::Widget(const wchar_t* initialString) :
        myValue(0)
{
        myValue = new char[strlen(initialString) + 1];
        strcpy(myValue, initialString);
}

Widget::~Widget()
{
        delete[] myValue;
}

int main()
{
        Widget w1;
        Widget w2 = w1;
        // When main exits, w1 will call its destructor. The =
        // has w2 pointing to the memory w1 has deallocated. When
        // w2 is destroyed it will try to deallocate this memory.
}
```

Example 41 “Safe” class having copy constructor, operator=, and destructor

```
class Widget
{
public:
        Widget(const wchar_t* initialString);
        ~Widget();

        Widget(const Widget& other);
        Widget(Widget&& other);
        Widget& operator=(const Widget& other);

        Widget& operator=(Widget&& other);

        // ...
private:
        wchar_t* myValue;
        // ...
};

Widget::Widget(const wchar_t* initialString) :
        myValue(new wchar_t[wcslen(initialString) + 1])
{
        wcscpy(myValue, initialString);
}

Widget::~Widget()

{
        delete[] myValue;
}

Widget::Widget(const Widget& other) :
        myValue(new wchar_t[wcslen(other.myValue) + 1])
{
        wcscpy(myValue, other.myValue);
}
```

```
Widget::Widget(Widget&& other) :
        myValue(other.myValue)
{
        other.myValue = nullptr;
}

Widget& Widget::operator=(const Widget& other)
{
        if ( this != &other )
        {
                delete[] myValue;
                myValue = new wchar_t[wcslen(other.myValue) + 1];
                wcscpy(myValue, other.myValue);
        }
        return *this;
}

Widget& Widget::operator=(Widget&& other)
{
        if ( this != &other )
        {
                delete[] myValue;
                myValue = other.myValue;
                other.myValue = nullptr;
        }
        return *this;
}
```

[Henr: 5.11, Mey1: Item 11, 27, 45]

Rule 69¤ **Do not overload operators new and/or delete.**

Rule 70 ¤ **Initialize all data members in a constructor in the order that they were declared. C++ 11: Use header initialization.**

A member can only be left out of the initialization list when it is of class type and its' default constructor is desired.

Class data members (i.e. data members that are themselves instances of some class) must be initialized in the member initialization list, not by assignment within the constructor body. Failure to follow this rule can cause run-time inefficiencies to occur.

To understand this rule requires understanding the construction process:

1. Storage for the object is allocated (either by the compiler in the static data area or on the stack, or dynamically by the program on the free store).
2. If the class is derived from other classes then the construction process is recursively applied to each base class in the order in which they are declared. The base class constructors are called automatically by the compiler but, by using a member initialization list, the programmer can control which constructor for each base class is called.
3. The construction process is recursively applied to each class data member in the order in which they are declared. This happens automatically but, by using a member initialization list, the programmer can control which constructor for each class member is called.

4. Finally, the body of the constructor is executed.

As can be seen from the above description class data members are ***always*** initialized prior to the execution of the constructor body. If the class data member is subsequently assigned to, then the initialization was (probably) wasted. Also note that the class data members are initialized in the order in which they are declared in the class definition, ***not*** in the order in which they appear in the initialization list.

If a class data member does not appear in the member initialization list then its default constructor will be called. (In this case if the default constructor does not exist then the compiler will signal an error.) If a class data member appears in the member initialization list then the appropriate constructor can be called and assignment within the class body should not be necessary.

Example 42 Initialize members in the order they were declared

```
Image::Image(
        const ScanList& initialScanList,
        const Viewport& initialViewport) :
        myScanList(initialScanList),
        myViewport(initialViewport)
{
        myViewport.centerX();
}
```

[Henr: 5.5, 5.6; Mey1: Items 12, 13]

In C++ 11, prefer initializing variables in the header directly as opposed to initialization in the constructors. This will ensure that default initialization of values will always occur and we only have to worry about initializing non-default values in our constructors.

As we add more constructors, this will also keep code duplication at a minimum by reducing redundant initializations. It will also prevent us from forgetting to default initialize some members.

Example 43 C++11: Use header initialization

```
class MyClass

{
public:
        MyClass(const wchar_t* initialString);

private:
        wchar_t* myPrivateString {L"DefaultValue"};
        int myValue {2};

        // Default-initilization, empty string
        std::string myString {};

};
```

5.15 Inheritance

Inheritance is a feature in C++ that can easily be abused. Good heuristics are to use it sparingly and only when there is a clear benefit. Do not make classes base classes because you “might” extend them in the future. Avoid multiple inheritance unless there is a clear benefit to using it. Really deep inheritance trees should be avoided.

Rule 71 ¤ A base class shall define a virtual destructor.

An instance of a derived class that is accessed through a pointer or a reference to its base class will be destroyed by the base class destructor and not by the derived class destructor if the base class destructor is non-virtual. In this case the derived class destructor is never called. If the base class destructor is virtual then both the base class and derived class destructors are called.

[Henr: 10.4, Mey1: Item 14]

Rule 72 ¤ An inherited default parameter value shall not be redefined.

Inherited virtual functions are dynamically bound, while default parameter values are statically bound. This means that a virtual function defined in a derived class with a redefined default parameter value will actually use the default parameter value from the base class.

[Mey1: Item 38]

Rule 73 ¤ Avoid calling virtual functions in a base class' constructor.

You must know what you are doing if you invoke virtual functions from a constructor in a base class. 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 44 Override of virtual functions does not work in the base class' constructors

```
class Base
{
public:
       Base(); // Default constructor
       virtual void foo();
};

Base::Base()
{
       foo(); // Base::foo() is always called.
}

// Derived class overrides foo()
class Derived : public Base
{
public:
       virtual void foo();
       // foo is overridden
};

int main()
{
       // Base::foo() called when the Base-part of Derived is constructed.
       Derived fooClass;
}
```

Rule 74 Do not use inheritance for "has-a" relations. Prefer aggregation or composition to inheritance.

Use inheritance when you want to express "isa" relations, e.g. a Manager "isa" Person. An Employee "isa" person. An Employee "has-a" employee id number and company telephone number. It would be correct for Employee to inherit from the Person class, but it would be incorrect for Employee to inherit from the IDNumber class or the CompanyTelephoneNumber class.

[Mey1: Items 35, 40]

In many cases, classes can be designed to implement a set of functionality using inheritance *or* aggregation. In these cases, aggregation (or its stronger form, composition) is preferred. The inheritance relationship is the strongest form of coupling in C++ and this coupling needs to be taken into account. Aggregation offers less coupling and in many cases is the desired alternative.

[Sutt: Item 24]

Rule 75 Pointers or references to derived classes should always be able to be used where a pointer or reference to corresponding base class is used.

This is a statement of the Liskov Substitution Principle which can be paraphrased as "Subtypes must be substitutable for their base types." [5]. Public inheritance should follow the IS-A relationship (a derived type IS-A base). This IS-A relationship pertains to the expected behavior of the base class. If a pointer to a base class contains a derived object that does not have the expected behavior of the base class, then it is not substitutable and client code will likely misuse it.

[Henr: 10.8; Mart: pp. 111-125]

Rule 76 Use Multiple Inheritance of implementation judiciously.

This applies in any cases where the classes being inherited are not pure interface classes. Choosing to use multiple inheritance in C++ has the potential to substantially increase program complexity as compared to single inheritance. Understand the related issues and read the reference below before venturing into this area.

[Mey1: Item 43]

5.16 Objects

Rule 77 Objects that use `new` to create other objects should be responsible for their destruction. C++ 11: Use `std::unique_ptr`, `std::shared_ptr`, or `std::weak_ptr` instead of raw pointers.

C++ doesn't have garbage collection and so it is imperative that we manage our own objects. In addition, adhering to this rule can also point out flaws in one's architecture. An exception to this rule is Factory-like [2] objects which create objects and then hand off their ownership to another object.

C++ 11: Smart pointers provide exception safety mechanisms by destroying resources at a scope level. In this case, the smart pointers will be responsible for the lifetime of the resource.

Rule 78 ¤ **If a raw pointer is required use new and delete instead of malloc, calloc, realloc and free.**

The C functions malloc, calloc and free can cause conflicts with the use of the C++ new and delete operators. It is dangerous to invoke delete for a pointer obtained via malloc, and to invoke free for anything allocated using new because new and delete call constructors and destructors, while malloc and free do not.

[Henr: 13.1; Mey1: Item 3]

Rule 79 ¤ **Do not delete `this`.**

Manipulations of the `this` pointer should be avoided at all costs.

[Henr: 8.4, Mey1: Item 3]

Rule 80 ¤ **Avoid passing the `this` pointer to another object through the member initializer list of a constructor.**

Passing the this pointer to another object through the member initializer list can be dangerous as the current object might not be fully constructed.

Example 45 **Dangerous use of this in member initializer list**

```
class Parent;

class Child
{
public:
       explicit Child( Parent* parent );

       // Add class definition
};

class Parent
{
public:
       Parent() :
              child( this )
{
}

// Add class definition and initialize members

private:

       // You can access these members from the Child class constructor.
       std::uint8_t age;
       std::string location;

       Child child;

       // Don't try to access this member in the Child class constructor.
       // It will be initialized eventually, but not until after parent is
       // fully constructed.
       //
       // The this pointer in the constructor of Parent will provide a
       // reference for later, not immediate, usage.
```

```
        std::size_t income;
};
```

It is safe to use the this pointer as long as no uninitialized members or virtual functions are used. An object will initialize its members according to the order in its definition (header typically). Since this requires a high degree of carefulness, it is not recommended to pass the this pointer as such.

Rule 81 Avoid the creation of temporary objects.

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. The compiler should warn of these occurrences.

Rule 82 Avoid interdependencies between global objects.

The C++ Standard states that the calling order of constructors for global objects defined in separate source files is undefined. For this reason, it is unwise for a global object to depend on other global objects. For global (and static) objects defined in a single source file, the C++ Standard guarantees that the constructors will be called in order of definition.

Example 46 Dangerous use of static objects in constructors

```
// Hen.h
class Egg;
class Hen
{
        Hen();
        ~Hen();
        // ...

        void makeNewHen(Egg*);
        // ...
};

// Egg.h
class Egg {};
extern Egg theFirstEgg;

// FirstHen.h
class FirstHen : public Hen
{
public:
        FirstHen();
        // ...
};
extern FirstHen theFirstHen;

// FirstHen.cpp
FirstHen theFirstHen; 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);
}
```

[Mey1: Item 47]

Rule 83 **Avoid global and static data.**

Global and static data can cause several problems. In an environment where parallel threads execute simultaneously, they can make the behavior of code unpredictable because global or static data is not reentrant. Code that depends on global data can be very difficult to debug.

5.17 Error Handling

Rule 84 ¤ **Prefer exceptions over fault codes whenever possible.**

Two important characteristics of a robust system are that all faults are reported and, if the fault is so serious that continued execution is not possible, the process is terminated. In this way, the propagation of faults through the system is avoided. It is better to have a process halt itself than to spread erroneous information to other processes. In achieving this goal, it is important to test fault codes from library functions. The opening or closing of files may fail, allocation of data may fail, etc. One test too many is better than one test too few.

[Henr: 12.1]

Rule 85 ¤ **Avoid using assert in product code.**

Assertions are only enabled in code that is compiled in debug mode so they are useful only for detecting errors during testing. When compiled in release mode, the asserted condition goes unchecked which can result in difficult to diagnose errors. Prefer to throw an exception.

Rule 85a ¤ **C++11: Prefer using static_assert.**

Static_asserts are evaluated at compile time. The compiler evaluates the expression and writes the specified string as an error message if the expression is false.

Example 47 **static_assert**

```
static_assert(sizeof(long) >= 8, "64 - bits required");

struct S { X m1; Y m2; };
static_assert(
        sizeof(S) == sizeof(X) + sizeof(Y),
        "unexpected padding of S");
```

Rule 86 ¤ **Resources shall be properly released even in the presence of exceptions.**

This rule is known as the “Basic Exception Safety Guarantee”. A software developer should consider exception handling seriously. Exception safety is not just an extra feature to add later, but a crucial aspect of class design. Resources do not only include memory resources, but any resource that is acquired (e.g., database locks, handle objects, critical sections, etc.).

[Sutt: Item 11]

Rule 87 ¤ **A function that isn't going to handle an exception shall propagate the exception to the caller.**

This is known as exception neutrality. If the function can't properly handle an exception it should not prevent the exception from being propagated to higher levels where it could be handled. This does not apply to cases where it is necessary to translate an exception in a final handler (e.g., if you're in the final handler of a thread and you need to translate the exception to a system event to be handled by another part of the system).

[Sutt: Item 8]

Rule 88 ¤ **Destructors shall neither throw nor allow exceptions to propagate. C++11: Mark destructors with the noexcept qualifier.**

The C++ Standard states that any code that allocates or deallocates an array of objects whose destructors could throw can result in undefined behavior. Destructors that can allow an exception to escape have the result that neither new[] nor delete[] can be made exception-safe. In addition, if a destructor throws while an exception is active (i.e., during stack unwinding) the associated process will be terminated. Therefore, destructors and deallocation functions should be written as if they had an exception specification of "`throw()`". In C++ 11, use the `noexcept` qualifier on destructors instead of `throw()`.

[Sutt: Item 16; Mey2: Item 11]

5.18 Style

The layout and formatting of our code can have a greater impact on readability than most engineers give them credit for. The rules for layout and formatting that we have set out here are based on typography and layout practices in use by graphic artists, as well as programming conventions used in the industry. The principal goal of this style section is *consistenc*y. A number of the rules are somewhat arbitrary, i.e. indent with 4 spaces as opposed to, say, 2. We could have chosen either indentation. It's not important which one we chose, it's important that we made a choice. Studies [14] have shown that consistent style makes a measurable difference to the expert programmer. For an excellent discussion of the merits of layout and style, please refer to Chapter 18 of [9].

Rule 89 **Use consistent formatting and style within a package or subsystem.**

There are many areas of style that are not covered by the rules in this section. The programmer is a liberty to make some style choices, however, the style choices made in a package or subsystem should be consistent for readability and ease of maintenance.

Rule 90 **Indent using four spaces.**

Code should be indented by four spaces beyond the enclosing structure. Studies have shown that indentation of 2 to 4 spaces is optimal [3]. We have chosen 4 as our standard. Exceptions are the `public`, `protected`, and `private` keywords which do not require any indentation.

Example 48 Code indentation

```
int main()
{

int value = 0;
while (value < firstBound)
{
       // code indentation is 4 spaces
       if (value < secondBound)
```

```
        {
                sendValue(value);
        }

        value += 2;
}
```

[McCo: pp. 409-410]

Rule 91 Don't use tabs in code.

Using tabs with your editor set to tabs=4 is *not* indenting 4 spaces. The use of tabs will cause erratic behavior in other tools used to view the code. For this reason, tabs are prohibited.

Rule 92 Put only one statement per line

This aids clarity, and, more importantly, aids debugging of the statements. The expressions comprising a `for` loop are an exception.

Rule 93 Put open and close braces on lines by themselves, indented to the same level as the code before the block (i.e., the `if`, `for`, etc. statement).

The opening and closing braces surrounding a block each go on their own line, indented at the same level as the code above the block. This applies to structure and class declarations as well as loops. Initialization of structures should, in general, be formatted the same way. However, when the structure is very small, the initialization can occur on the same line as in the last example below.

Example 49 Correct brace placement

```
if (condition)
{
        // code indentation is 4 spaces
        ...;
}
else
{
        // else clause code here
        ...;
}
```

Example 50 Correct struct declaration

```
class AnyClass
{
public:
        AnyClass();
        ~AnyClass();

private:
        int myAnyValue;
};

struct Foo
{
        int  amount;
        char label;
};
```

Example 51 Initialization of a small structure

```
Foo someFoo = { maxAmount, 'X' };
```

Rule 94 If the function's parameter list exceeds the character column of 120, place all parameters on separate lines.

The emphasis is on consistency within a class or module. Pick an acceptable way to follow this rule (see examples below for some ideas), and stick to it.

Example 52 Correct function definition with long parameter list

```
Element* createElement(
        long int elementID,
        double temperature,
        const char* label,
        const char* filename)
        {
            // ...
```

Example 53 Alternate function definition with long parameter list

```
Element* Element::setElement(
        long int elementID,
        double temperature,
        const char* label)
{
        // ...
```

Rule 95 Use whitespace liberally.

Don't be afraid to add white space to clarify the visual presentation of your code.

Example 54 Use white space to clarify code

```
auto foo = barFunction(cat, dog); // Right

for ( auto x : elements )
{
       …
} // Also right

foo = barFunction(cat, dog); // Not so right.
```

[McCo: p. 408-409]

Rule 96 Use parentheses to clarify the order of evaluation for operators in expressions.

There are a number of common pitfalls having to do with the order of evaluation for operators in an expression. Binary operators in C++ have associativity (either leftward or rightward) and precedence. If you have any doubt about the order of evaluation in an expression, use parenthesis to explicitly tell the compiler what you want. C++ allows the overloading of operators, something which can easily become confusing. For example, the operators << (shift left) and >> (shift right) are often used for input and output. Since these were originally bit operations, it is necessary that they have higher priority than relational operators. This means that parentheses must be used when outputting the values of logical expressions.

Example 55 Problem with the order of evaluation

```
// Interpreted as (a < b) < c, not (a < b) && (b < c)
if ( a < b < c )
{
       // ...
}

// Interpreted as a & (b < 8), not (a & b) < 8
if ( a & b < 8 )
{
       // ...
}
```

Example 56 When parentheses are recommended

```
int i = a >= b && c < d && e + f <= g + h; // Wrong.
int j = (a >= b) && (c < d) && ((e + f) <= (g + h)); // Right.
```

5.19 The C++ Standard Library

Rule 97 Prefer using the C++ Standard Library when there is a choice.

If you have the choice between using a facility provided by the C++ Standard Library and a home grown version, use the C++ Standard Library. This will aid in making the code more maintainable, extendible, and efficient. This assumes that you have an understanding of what's in the Standard Library.

Rule 98 Prefer C++ Standard Library containers to C style arrays or customized containers.

The Standard containers are more robust and flexible and provide better mechanisms to support exception safety than built-in arrays such as C style arrays. If you do create a custom data structure or algorithm, they must be compatible with the standard algorithms and containers. The custom container must have properly defined iterators for use in container-agnostic algorithms. The iterator classes shall not inherit from std::iterator as it is deprecated in C++17.

A good reference for standard conforming iterators can be found at the following resource: http://en.cppreference.com/w/cpp/concept/Iterator

Rule 99 Prefer Standard algorithm calls to hand-written loops when using Standard containers.

C++ Standard algorithms are often more efficient than the loops programmers produce. Writing loops is more subject to errors than calling Standard algorithms. Standard algorithm classes often yield code that is clearer and more straightforward than the corresponding explicit loops.

Example 57 Prefer Standard Algorithms to hand-written loops

```
std::list<int> intList;

// do something with the list
// Want the sum of all elements in the list

int sum = 0;
```

```
// Avoid finding sum using hand-written loops
for (
        std::list<int>::iterator iter = intList.begin();
        iter != intList.end();
        ++iter)
{
        sum += *iter;
}

// C++11
// Prefer algorithm call using the accumulate() algorithm,
// with auto type deduction and non-member iterator
// functions (std::begin, std::end, etc.) as follows:
const auto betterSum =
        std::accumulate(
                std::begin(intList),
                std::end(intList),
                0);
```

[Mey3: Item 43]

Complete list of standard algorithms: http://en.cppreference.com/w/cpp/algorithm

Rule 100 Prefer unicode strings to single byte strings

Unicode strings (e.g., std::wstring) are not constrained in character set as are ANSI or single byte strings. This makes wide strings preferable when writing applications that can be used in multiple language settings.

Rule 101 Encapsulate raw pointers in smart pointer memory management classes.

Using a smart pointers such as std::auto_ptr will help prevent memory leaks and provide exception-safe memory management. Note: that due to the transfer of ownership on copy or assignment, auto_ptr's are not compatible with Standard containers. If you want to put managed pointers into Standard containers you will need to develop your own smart pointers.

Example 58 Using smart pointers

```
// An example of how to use a home grown smart pointer, Su::SharedPtr
std::vector<SharedPtr<int>> someVector;

for ( int idx = 0; idx < 20; ++idx )
{
        someVector.push_back(SharedPtr<int>(new int(idx)));
}
```

Rule 99a C++11: Always use the new standard smart pointers and non-owning raw pointers. Never use owning raw pointers.In C++11 std::auto_ptr is deprecated. Use the new smart pointer types.

Use unique_ptr to hold a pointer to an object that has a single owner; use shared_ptr when there are potentially many owners; and use weak_ptr to reference an object that is owned by a shared_ptr and you may want to assume temporary ownership. (The weak_ptr must be converted to a shared_ptr to assume temporary ownership.)

Use non-owning raw pointers to reference objects that are guaranteed to outlive the referencing object. Never use owning raw pointers and delete. Exceptions occurring between new and delete will result in a leak.

5.20 Parts of C++ to Avoid

Rule 102 Avoid C-style I/O: use the `iostream` library instead.

C-style I/O is not type-safe nor extensible, while iostream is.

[Henr: 13.2, Mey1: Item 2]

Rule 103 ¤ Do not use `setjmp()`, `longjmp()` or `goto`.

`setjmp()` and `longjmp()` do not properly unwind the stack and so local objects will not be destroyed when they should be.

`goto` breaks the control flow and can lead to code that is difficult to comprehend. In addition, there are limitations for when `goto` can be used. For example, it is not permitted to jump past a statement that initializes a local object having a destructor.

[Henr: 13.3]

Rule 104 Do not use unions of classes.

Unions make assumptions about the memory footprint of the types they contain. This will not work for classes, and is pretty tricky for built-in types as well.

[Henr: 13.7]

Rule 105 Prefer references to pointers whenever possible.

Pointers can point to null, making them tedious with null comparisons and error-prone without. On the other hand, references cannot point to null.

This rule also covers the usage of `dynamic_cast`, which works with both pointers and references. A `dynamic_cast` to a pointer will return null if the cast fails. If we are not going to handle the invalid pointer or if we are going to handle it generically by throwing `std::bad_cast`, prefer to use a `dynamic_cast` to a reference instead. This will ensure the validity of the return by automatically throwing `std::bad_cast` if the cast has failed.

C++ 11: In general follow these guidelines for passing params that represent dynamic memory.

4) for read only access pass by `const T*` or `const T&`.

5) for transferring ownership pass `std::unique_ptr<T>` or `std::shared_ptr<T>` by value.

6) for conditionally transferring ownership pass `std::unique_ptr<T>` or `std::shared_ptr<T>` by r-value reference – should be used sparingly

Example 59 Passing dynamic resources

```
void readOnly(
       const T* p1,
       const T& p2);
```

```
void own(
        std::unique_ptr<T> uPtr,
        std::shared_ptr<T> sPtr);

void conditionallyOwn(
        std::unique_ptr<T>&& uPtr,
        std::shared_ptr<T>&& sPtr);

int main()
{
        auto uPtr = std::make_unique<T>();
        auto sPtr = std::make_shared<T>();

        readOnly(uPtr.get(), *uPtr);

        // uPtr is now moved to the param
        // and points to NULL, sPtr is copied by value and
        // ownership is now shared between main() and own().
        own(std::move(uPtr), sPtr);

        // Decisions about ownership are now delegated to
        // conditionallyOwn(). Use sparingly
        conditionallyOwn(std::move(uPtr), std::move(sPtr));

}
```

Rule 106 C++98: Pointers shall not be compared to NULL or assigned NULL; the value 0 shall be used instead.

NULL is defined either as (void *) 0 or 0 in the ANSI-C Standard. If NULL is (void *) 0, it cannot be assigned an arbitrary pointer without an explicit type conversion.

Example 60 Different comparisons of pointers

```
wchar_t* bufferP = new wchar_t[100];

if ( bufferP == 0 ) // Right.
{
        std::wcout << L"New failed." << std::endl;
}

if ( !bufferP ) // Preferred.
{
        std::wcout << L"New failed." << std::endl;
}

if ( bufferP == NULL ) // Wrong.
{
        std::wcout << L"New failed." << std::endl;
}
```

Rule 106a C++11: Always use nullptr for a null pointer value rather than the macro NULL or the literal 0.

5.21 C++11 Features

Rule 107 Use auto.

The auto keyword should be used where the true type is difficult for the programmer to determine, such as the types of most lambda functions.

Example 61 Using auto

```
This example taken from an article "Elements of Modern C++ Style"
by Herb Sutter[19]

C++98:
       std::binder2nd<std::greater<int>> x =
              std::bind2nd(std::greater<int>(), 42);

C++11: auto x = [](int i) { return i > 42; };
```

The auto keyword will make some typedefs unnecessary.

Rule 108 Prefer the non-member functions std::begin(), std::end(), std::cbegin(), std::cend(), std::rbegin(), std::rend(), std::crbegin(), and std::crend().

Use the non-member begin() and end() (rather than x.begin() and x.end()) because they are extensible and can be adapted to work with arrays as well as STL-style collections.

If possible, prefer to use const iterators (std::cbegin(), std::cend(), etc.) to non-const iterators (std::begin(), std::end(), etc.).

Write non-member begin() and end() overloads to adapt non-STL collection types for traversal using the STL coding style,

Example 62 Non-member begin() and end()

```
This example taken from an article "Elements of Modern C++ Style"
by Herb Sutter[19]

std::vector< int> v;
int a[100];

// C++98:
std::sort(v.begin(), v.end());
std::sort(&a[0], &a[0] + sizeof(a) / sizeof(a[0]));

// C++11:
std::sort(std::begin(v), std::end(v));
std::sort(std::begin(a), std::end(a));
```

Rule 109 Prefer lambda functions whenever appropriate.

Lambda functions offer a concise and easy to understand alternative to function pointers and function objects. Like function objects, lambdas can maintain state. Unlike function objects, lambdas are defined in-line making them easier to read. Lambdas also have access to variables available in their enclosing scope. Lambdas are compatible with existing STL algorithms and newer libraries are likely to require their use.

Good practices in the usage of lambdas include:

- Specifying the capture type, that is, by value [=] or by reference [&].

- Using few parameters. Lambda functions should generally be small and concise.
- Keep the definition of a lambda function small.

Example 63 Usages of lambda functions

```
std::array< int, 1000 > container;

std::iota(
        std::begin(container),
        std::end(container),
        0);

auto count = std::count_if(
        std::cbegin(container),
        std::cend(container),
        // The 3rd argument here is a lambda
        [=](auto item)
{
        // Find items divisible by 5
        return (item % 5) == 0;
});
```

Rule 110 Support move semantics.

Rvalue references were added in C++11 and are used primarily to support the efficient transfer of ownership of resources from a temporary object without having to copy that resource.

See item 24 in Scott Meyers' Effective Modern C++ book.

.

Example 64 Class designed with move semantics

```
class CustomResource
{
public:
        CustomResource(std::size_t newSize) :
                buffer(new char[newSize]),
                size(newSize)
        {
        }

        ~CustomResource() noexcept
        {
                delete[] this->buffer;
        }

        CustomResource(const CustomResource& other) = default;
        CustomResource& operator=(const CustomResource& rhs) = default;

        CustomResource(CustomResource&& other) :
                buffer(other.buffer),
                size(other.size)
        {
                other.buffer = nullptr;
                other.size = 0;
        }
```

```
		CustomResource& operator=(CustomResource&& rhs)
		{
			if (this != &rhs)
			{
				delete[] this->buffer;

				this->buffer = rhs.buffer;
				this->size = rhs.size;

				rhs.buffer = nullptr;
				rhs.size = 0;
			}

			return *this;
		}

		std::vector< CustomResource > generateResourcesGood(
			std::size_t count)
		{
			std::vector< CustomResource > resources(count);

			// [Named] Return Value Optimization
			return resources;
		}

		std::vector< CustomResource > generateResourcesBad(
			std::size_t count)
		{
			std::vector< CustomResource > resources(count);

			return std::move(resources);
		}

	private:
		char* buffer;
		std::size_t size;
	};
```

Good practices in the usage of move semantics include:

- Do not return rvalue references to temporaries. In other words, do not 'move' them out of a function as part of its return. This will impede a compiler optimization called return value optimization (RVO), which will completely elide any object creation. As a result, it is perfectly safe to return the temporary directly. In the case where RVO is not supported, the temporary will automatically be converted to an rvalue reference (moved). Forcing the conversion can only hurt the compiler, not help.
- Assume move operations are not cheap and be careful with moving objects or buffers allocated on the stack. The stack controls the lifetime of the allocation, leaving us without the option to prevent destruction of certain objects. As a result, moving buffers from the stack occurs for every single element, effectively making it as slow as a copy.

Rule 111 Use the uniform initialization syntax and initializer lists

Prefer using {} braces and an initializer list to initialize objects of POD (Plain Old Data) types, and for passing function arguments and returning values without having to explicity name the type.

Example 65 Uniform initialization

```
This example taken from an article "Elements of Modern C++ Style"
by Herb Sutter[19]

// C++98:

rectangle w(origin(), extents());
// oops, declares a function, if origin and extents are types

std::complex<double> c(2.71828, 3.14159);
int a[] = { 1, 2, 3, 4 };

std::vector<int> v;
for ( int i = 1; i <= 4; ++i )
{
       v.push_back(i);
}

// C++11:

rectangle w  { origin(), extents() };
std::complex<double> c { 2.71828, 3.14159 };

int a[] { 1, 2, 3, 4 };
std::vector<int> v { 1, 2, 3, 4 };
```

This initializer syntax for POD type is now the same as the familiar array initializer syntax. Using the {} initializer syntax for arguments and returns avoids having to specify the redundant typename.

Rule 112 Use ranged for loops whenever possible.

Ranged for loops allow one to perform container agnostic iteration without needing to use a counter variable. The counter variable adds more noise to the code and can even cause bugs if the counter is not changed appropriately.

Example 66 Ranged For

```
std::array< BigObject, 5 > myArray { ... };

// uses auto to retrieve a local copy of every iterated object.
for (auto elem : myArray)
{
	...
}

// uses auto& to prevent the copy constructor from being called
// and allow for modification.
for (auto& elem : myArray)
{
	...
}

// uses const auto& to prevent the copy constructor from being
// called and prevent modification.
for (const auto& elem : myArray)
{
	...
}

// uses auto&& to prevent either the copy or move constructor
// from being called and to iterate over rvalue references or
// temporaries.
for (auto&& elem : myArray)
{
	...
}
```

Rule 113 Use C++ standard threading constructs whenever available.

Standard threading concepts are more user-friendly, portable, and prevent many common mistakes compared to directly calling the OS functions. They are also exception safe by design.

Example 67 C++ std::thread example

```
// Wrong

DWORD WINAPI threadFunc(LPVOID lpParam)
{
	...
}

int main()
{
	int param = 5;

	HANDLE threadHandle =
		CreateThread(
                    NULL,
                    0,
                    threadFunc,
                    &param,
                    0,
                    NULL);
}

// Right

int threadFunc(int arg)
{
       ...
}

int main()
{
       std::thread(threadFunc, 5);
}
```

**Example 68		C++ std::mutex example**

```
// Better

HANDLE mutex =

       CreateMutex(
              NULL,
              FALSE,
              NULL);

// Do something with mutex

CRITICAL_SECTION cs;
InitializeCriticalSection(&cs);

void compute()
{
       EnterCriticalSection(&cs);

       // Do something important
       LeaveCriticalSection(&cs);

}


// Better
std::mutex mtx;

void compute()
{
       // mtx is released when lock_guard goes out of scope
       std::lock_guard< std::mutex > lock(mtx);

}
```

**Rule 114	Prefer task based to thread based programming.**

Task based programming provides a higher level of abstraction and frees the programmer from the details of thread management, instead pushing this to the OS. Task based programming also allows us to use a return value, and provides a simple exception handling mechanism.

Example 69 Task Based Programming

```
int threadFunc(int arg)
{
        ...
}

int main()
{
        // OK, per prior rule, but perhaps not preferred.
        std::thread(threadFunc, 5);

        // Preferred
        // fut is of type std::future
        auto fut = std::async(threadFunc, 5);

        // The default above allows the OS to determine whether the task
        // is performed asynchronously or deferred. This can be specified
        // manually where asynch will force the task to run on different
        // thread, whereas deferred will not run until a call to either
        // std::future::get()or std::future::wait() is called. If neither is
        // called, the task will never be executed.
        auto fut2 = std::async(std::launch::async, threadFunc, 5);
        auto fut3 = std::async(std::launch::deferred, threadFunc, 5);
        fut3.get();
}
```

SAP-EDM Signature Information
- generated automatically by SAP system **P41** -

Appendix to Document: **10011642 QMS 001 03 , ECO: 660751**
Sheet generated at : **2017-07-12T16:08:32-02:00**
Originator **: SIEMENS Healthcare, P41**
Signatures related to this document and performed in SAP:

Meaning	system date and time	surname, given name of signee
AUTHOR	**2017-07-11T23:18:45-02:00**	**Burke, Laurel**
APPROVAL	**2017-07-12T16:07:58-02:00**	**Burke, Laurel**