C++ Coding Standard

1 Outline

BMS Document	**WI1003**
Confluence Version	31
Purpose	To define C++ code development rules and guidelines
Scope	Engineering - PRP
Owner	Engineering Software Manager

2 Work Activity

INTRODUCTION

Purpose

The purpose of this coding standard is to provide a common set of rules and recommendations for developing code in C++. Adherence to this standard should result in code which is easier to comprehend and to maintain, and which is more robust and portable.
Everyone will have their own ideas on what constitutes the best clarity and coding practices. However there is benefit in having a consistent set of rules and guidelines that everyone in a project follows regardless of their own individual preferences. Where there is disagreement, it is the intention that this will simply be down to a matter of taste rather than any technical deficiency of the standard.
So through adoption of these conventions, putting aside personal preference, we become accustomed to the house style and achieve greater collective clarity than each using our own preferences.

Scope

This document specifies the Raymarine coding standard for C++ with regard to naming conventions, code layout and general coding practices.
There is no attempt here to address the subject of design methodology or development process.
It is intended that this document serve the purposes of the software effort. If in a given situation it is appropriate to impose certain areas of these standards less rigorously, for example for the sake of efficiency in a critical routine or if a particular tool makes it necessary, then this is considered acceptable. However the basic principles of good coding must be adhered to. If any of the rules are broken, then the relevant code should be accompanied by an explanatory comment.

Using this document

This document contains both rules which must be followed in order to write reliable code, as well as hints and guidelines which have been found to be useful and good practice through experience.
!worddavc57d1e2f3f3d2eb4286dbe901e60a143.png|height=30,width=33!The items which are mandated are marked with an exclamation mark icon (as this line is) and any deviations to these points will need to be individually explained at code review. Since reviewing code is not an easy process it is expected that deviations from these rules will be rare.

Language Subset

Raymarine use ISO/IEC 9899:1998 C++

Every project should use the smallest possible subset of ISO C++

In particular a project policy should be defined for use of:

- RTTI.
- IO Streams
- Exceptions
- STL

A file should not exceed 1000 lines

- any files over 2000 lines would need strong justification

A function should be no more than 100 lines long

- Approximately 50 lines per function is recommended
- More than 200 lines would need strong justification

Line Length

- Lines should be short to facilitate reading (8-10 words per line)
- Lines should not exceed 120 characters.
- Consider breaking up long statements into simpler ones, using local variables if required. (A good compiler optimizer will recombine your code & lose the locals.)

A function should contain no more than 5 levels of indentation

Justification

- *Length of functions is a common cause of errors and leads to difficulty in debugging and maintaining code. The figures of 50 & 100 are designed to put a number to the 'about a screen' commonly accepted guidelines.*

One Statement Per Line

- There should be only one statement per line

Don't use comma operator, use separate statements.

- The comma operator should not be used

Use the conditional expression ?: with caution

- Consider whether the intent can be more clearly expressed with an if else or other construct
- Do not nest ?: operators.
- It is often useful in simple initialisations and return statements.
- It is occasionally unavoidable e.g. in MACROs
- Put the condition in parentheses so as to set it apart from other code.
- If possible, the actions for the test should be simple functions or constants.

Example

Var = (*<condition>*) ? funct1() : func2();

or

// Where possible ?: statements should be transformed to:-

if (*<condition>*)

{

Var = funct1();

}

else

```
{

Var = funct2();

}
```

Use of goto, continue and break

- Remember that these add to complexity and simple structures should be used if possible
- *goto* statements should not be used.
- *break* and *continue* statements can usefully be employed to break out of switch, for, and while nesting with a success/failure return code.
- Comment each break and continue with a reference to the structure being exited

```
while (...)

{

...

if (disaster)

{

ErrorFlag = true;



break; // while (...)

}

}

...
```

```
if (ErrorFlag)

{

clean up the mess

}
```

One Variable Per Line

- Related to this is always define one variable per line:

Wrong:

```
char* A, B; // Looks like both A and B are char* s.
```

Right:

```
char* A = 0;

char B = 0;
```

Justification

- *Documentation can be added for the variable on the line.*
- *It's clear that the variables are initialised.*
- *Declarations are clear which reduces the probability of declaring just a char when you meant to declare a pointer.*

Use classes with private data only and use struct for public data.

Minimise scope

Declare entities in block, function or class scope rather than global scope and declare file-level globals (e.g. constants) as static, to prevent reference from outside.
Declare variables as late as possible, to show that they are not used before then.

Assertions

- Use a compile-time assert to enforce any compile-time-constant constraints.
- Use a run-time assert to enforce pre-conditions, post-conditions and invariants.
- Use a run-time assert to detect conditions that should be impossible.
- Do not use assertions for run-time error conditions.

Justification

- *Assertions help to detect errors in our assumptions / design errors*

Good Examples

```
#include "CompileTimeAssert.h"
typedef unsigned short int UINT16;
COMPILE_TIME_ASSERT(sizeof(UINT16) == 2);
```

Names

Make Names Fit

Names are at the heart of programming. If the name is appropriate everything fits together naturally, roles and relationships are clear, meaning is derivable, and operation will be as expected from normal reasoning.
Names must differ by more than case, i.e. don't declare a formal parameter of SpeedMetres and a local variable called speedMetres, because they are the same except in the case of the first letter, even if the latter starts by being initialised by the former:
Names should differ by more than one character, from anything that is accessible in the same scope.
Bad Example

```
void speedMetres (int SpeedMetres) // Bad choice of identifiers

{

...

}
```

Grammar

Use verbs for methods, nouns for members and classes, all spelt in UK English.
The following table shows the naming convention for each entity type:

Entity	Notes	Prefixes	Options	Style	Suffixes
Namespace	noun			CamelCase	
Typedef Name	noun			CamelCase	_t
Class Name	noun	C		CamelCase	
Template Name	noun	T		CamelCase	
Interface	noun	I		CamelCase	
Global Variable	noun		p	CamelCase	
Member Variable	noun	m_	r, p	CamelCase	
Argument	noun		r, p	CamelCase	
Local Variable	noun Note1		r, p	az_or_CamelCase	
Constants	noun (any scope)	c_	p	CamelCase	
Enum	noun	e_		CamelCase	
Struct / enum tag	noun			CamelCase	_t
Class Method	verb			CamelCase	
C style function	verb			lower_case	
Macro	noun or verb			ALL_CAPS	
Macro Parameters	noun			CamelCase	
Const ref. argument	noun			CamelCase	
Non-const ref arg.	noun	r		CamelCase	

Note1: Single letter variable names may be used for loop counters within small blocks only. For larger blocks and for all other variables, CamelCase must be used.
The styles are defined as follows:

Style	Description	Regex
CamelCase	First letter of each word is capitalised. Digits are allowed after the first character and also act as word separators. No underscores anywhere in these names.	([A-Z][a-z0-9]*)+
az_or_CamelCase	Single letter (small block only) or CamelCase	[a-z]([A-Z][a-z0-9]*)+
lower_case	All lower case, words separated by underscore. Digits are allowed after the first character and also act as word separators. No leading or trailing underscores.	[a-z](_?[a-z0-9])*
ALL_CAPS	All capitals, words separated by underscore Digits are allowed after the first character and also act as word separators. No leading or trailing underscores	[A-Z](_?[A-Z0-9])*

Class Names

- Name the class after what it is. If you can't think of what it is then that is a clue you have not thought through the design well enough.
- Compound names of over three words are a clue that your design may be confusing various entities in your system. Revisit your design.
- Avoid the temptation of bringing the name of the class from which a class is derived into the derived class's name. A class should stand on it's own, it doesn't matter what it derives from.
- First character in a class name is an upper case 'C', unless it is a template class or interface class, in which case the first character is an upper case 'T' or 'I' respectively.
- Note interface programming has been adopted as a language extension.
- Second character is upper case, in accordance with the word separation rule.
- No underscores ('_').

Good Examples

classCVesselPosition;

template <class T> class TRadarRenderer;

interfaceIDatabase;

Method and Function Names

- A method or function name should make clear what it does: CheckForErrors() instead of ErrorCheck(), DumpDataToFile() instead of DataFile(). Classes are often nouns. By making function names verbs and following other naming conventions programs can be read more naturally. This will also make functions and data objects more distinguishable.
- Use the same rule as for class names - upper case letters as word separators, lower case for the rest of a word.

Good Examples

bool CheckForErrors() const;

void DumpDataToFile();

- Use a preceding underscore for private functions if it's appropriate. This is optional.
- When you need to get rid of a warning for an unused parameter, comment the parameter out in the definition line if you can. Casting to void is acceptable if there is conditional compilation which means the parameter is sometimes used and sometimes not.

Class Attribute Names

- Attribute names should be prepended with the characters 'm_'.
- After the 'm_' use the same rules as for variable names.
- 'm_' always precedes other name modifiers like 'p' for pointer.

- Const reference parameters don't have an r prefix, but non-const references do. The justification for this is that a const reference isn't something the function can change, so it may as well have been passed by value semantically.

Justification

- Prepending 'm_' prevents any conflict with method names. Often your methods and attribute names will be similar, especially for accessors, although if you adhere to the method naming guidelines then the methods should be identifiable as verbs and the attributes as nouns.

Example

```
class CWaypointName

{

public:

typedef AUINT32 WaypointId_t; ///< Unique Waypoint Id

typedef std::string WaypointName_t; ///< Users Name for this Waypoint

WaypointId_T WaypointId() const; ///< Get the current Waypoint Id

private:

WaypointId_t m_WaypointId; ///< Waypoint Id, unique to system

WaypointName_t m_WaypointName; ///< Waypoint name, seen by User

}
```

Variable Names

Pointer Variables

- Never declare more than one variable on a line
- Pointers should be prepended by a 'p'

Justification

- The idea is that the difference between a pointer, object, and a reference to an object is important for understanding the code, especially in C++ where -> can be overloaded, and casting and copy semantics are important.
- Pointers really are a change of type so the '*' belongs near the type. Multiple declarations on a single line is banned so there is no ambiguity.

Example

```
String* pName = new String;

String* pName, Name, Address; // WRONG, only pName is a pointer.

String Name; ///< Comment each variable

String Address; ///< comment each variable
```

Reference Variables

- References should be prepended with 'r'.

Include Units in Names

- If a variable represents time, weight, or some other unit then include the unit of measure in the name so developers can more easily spot problems. For example:

```
AUINT32 m_TimeoutMsecs;

AUINT32 m_MyWeightLbs;
```

- In some cases it may be useful to turn units (i.e. engineering units) into a class so bad conversions can be caught.

Suffix Fixed point values with the scale factor used

- Variables with fixed point scaling should include the scale factor that has been applied in encoding the variable. For example:

AUINT16 DistanceMetresx4000 = 4000; // 1m

Constants

- All constants should be of the format 'c_*VariableName*', this applies whether global or member.
- This applies to constants only, there is no prefix indicating const-ness of arguments.

Justification

- Whilst the capitals and underscores (e.g. A_GLOBAL_CONSTANT) has been the format of choice for constants in the past, this new format has been chosen for improved readability and consistency with other class member variable naming conventions. The ALL CAPS format is reserved for MACROS.

Example

static const UINT32 c_GlobalConstant = 5;

Type Names

- When possible for types based on native types make a typedef.
- Typedef names should use the same naming policy as for a class with the suffix '_t' appended

Justification

- This is already standard practice and works well.
- The '_T' is appended to make it clear this is not a class.

Example

typedef UINT16 ModuleInfo_t;

typedef UINT32 SystemInfo_t;

- The practice of combining typedef and enum (or struct) is only required for dual C / C++ declarations. In this case the enum (or struct) tag should be omitted.

Example

```
typedef enum // For compatibility with C only

{

e_StateErr,

e_StateOpen,

e_StateRunning,

e_StateDying,

e_StateNumValues

} State_T;
```

#define and Macro Names

- The use of *#define* is discouraged, as stated in section 5.1
- Put #defines and macros in all upper case using '_' separators.

Justification

- *This makes it very clear that the value is not alterable and in the case of macros, makes it clear that you are using a construct that requires care.*
- *Some subtle errors can occur when macro names and enum labels use the same name.*

Example

```
#define SELECT_MAX(A,B) blah
```

```
#define IS_ERR(Err) blah
```

Enum Names

- Enumerated constants should take the form e_<TagName>UniqueName where <TagName> is common to the enum tag and to all items in the enumeration.
- When an enum is not embedded in a class, i.e. constant without class scoping, prefix the labels with a differentiating tag to prevent name clashes
- Use FirstCaps style for tags and enumerations
- It's often useful to be able to say an enum is not in any of its *valid* states. Make a label for an un-initialised or error state. Make it the first label if possible.
- Where the enum is a contiguous list a final item (e.g e_PinNumValues) is often useful to enable range-checking.

Example

```
enum Pin_t

{

e_PinUnknown,

e_PinOff, // might conflict with another OFF if Pin was not prepended

e_PinOn,

e_PinNumValues

} Pin_T;
```

C Style Function Names

- Most C functions will probably be incorporated into a project from legacy code, hence function names will already have been defined, however for new C functions use the GNU convention of all lower case letters with '_' as the word delimiter
- This naming applies to C++ functions that are global in scope as well as mixed compilation with C.
- When mixed compilation is required, extern C is required on the C++ declaration.

Justification

- *It makes C functions very different from any C++ related names.*

good Example

```
UINT32 some_function()

{

}
```

Namespaces

- Namespaces should be used to prevent class name conflicts among libraries from different vendors and groups.
- Use namespaces to scope definitions outside of classes.
- Never import namespace symbols ("using namespace") at global scope in a header file.
- Avoid "using" statements in implementation (.cpp) files.

Justification

- *Namespaces avoid conflicts between definitions with identical names in different parts of the code.*
- *When necessary, explicitly access a symbol with its fully qualified name ("MyNamespace::c_MyConst").*
- *Importing symbols at global scope in a header file may result in invisible conflicts that are hard to track.*

Good Example

```
namespace MyNamespace

{

typedef UINT32 MyIndex_t;

const UINT8 c_MyConst = 42;

};



UINT8 Foo = MyNamespace::c_MyConst;
```

Comments

Comments

- Good comments will explain why something is being done.
- What is being done is usually obvious, comments can help if it is unusual or not obvious.
- Use C-Style /* … */ comments for blocks
- Use C++ // comments for single lines.
- Use Doxygen with C style as defined in WI_05-12136.
- Use #ifdef <initials>_TO_DO to temporarily remove code and clean up before check-in.

Example

```
{

// Block1 (meaningful comment about Block1)

... some code


{

// Block2 (meaningful comment about Block2)

... some code

} // End Block2


} // End Block1
```

- Where a block of code needs to be commented-out, use pre-processor directives such as #ifdef <Initials>_TO_DO … #endif // <Initials>_TO_DO
- Where possible remove commented-out code before check-in.

Document Null Statements

- Always document a null body for a *for* or *while* statement so that it is clear that the null body is intentional and not missing code.

```
while ((*Dest++ = *Src++) != 0)

{

; // Everything is done in the condition

}
```

File Names

Naming Class Files

C++ File Extensions

- Use the *.h* extension for C++ header files and *.cpp* for C++ source files.

Class Definition in One File

- Each class definition should be in its own file where each file is named directly after the class's name.
- For a file containing a standard class definition the name of the class minus the initial 'C' should be used – e.g. for a file containing C<ClassName>, the filename would be '<ClassName>.h'.
- For files containing template definitions T<MyTemplate>, the filename should start with a 'T' e.g. 'T<MyTemplate>.h' and for files containing interface (abstract base) classes, the 'I' should be preserved, e.g I<MyInterface> is stored in 'I<MyInterface>.h'

Implementation in One File

- In general each class should be implemented in one source file:

ClassName.cpp

Include files

Include Files

- Every include file shall itself reference includes that it depends on so that the include file will compile cleanly even in isolation
- Forward references to classes shall be used to eliminate the need to include class declarations whenever possible
- Include references shall use the <system> or "project" notations appropriately
- Include files shall not reference absolute paths, in general they should be relative to a specific directory to minimise the number of include paths specified to the compiler
- Include files shall be treated as case sensitive (for portability to systems that are case sensitive)
- Include directory references shall use "/" rather than "\" for portability
- All function declarations should have named parameters, including Qt signals and slots, e.g. _Q_SIGNAL void ValueChanged(int Value)_ and not _Q_SIGNAL void ValueChanged(int){_}. The QObject::connection function needs these missing, but the metaobject compiler still handles these correctly.

Multiple Inclusion of Header Files

- Multiple inclusion protection shall always be incorporated.
- Protect by defining a pre-processor symbol constructed from *<FileName>*_H_INCLUDED.
- Use #ifndef instead of #if !defined() as it allows easier double-check that the define below matches the test.

#ifndef HEADERNAME_H_INCLUDED

#define HEADERNAME_H_INCLUDED

...

#endif // HEADERNAME_H_INCLUDED

Build Optimization

- Only include build include file optimization in a portable fashion and where it improves compilation speed.
- Note that #pragma once can cause problems on parallel build systems and may not give a significant improvement where the compiler includes an optimisation to recognise the multiple inclusion protection.

#ifndef HEADERNAME_H_INCLUDED

#define HEADERNAME_H_INCLUDED

#if defined(*_ghs) && defined(EDG_*)

#pragma once

#endif

...

#endif // HEADERNAME_H_INCLUDED

Base Types

Use domain specific typedef's to abstract away from standard types

- Standard types should be referenced with the following preference:

Use predefined types bool, size_t whenever appropriate.

In general use the new efficient 'at least' sized types

- These have at least the size of the sized corresponding type and are implemented in the most efficient size for the platform: AUINT8, AUINT16, ASINT32, ASINT64, etc.
- These are to be used after New E on MFDs, until then the fixed size types are to be used.

Use sized types when size is important

- e.g. in network comms, files or to ensure overflow will not occur.

- When size is important use Raymarine standard UINT8, UINT16, SINT32, SINT64, etc. from the appropriate global include file.

Justification

- *The size of standard types is compiler dependent. For portability we need to ensure that the size of types is defined where it matters.*

Initialise All Variables

Initialise all Variables

- You shall always initialise variables before use. Always. Every time.
- Initialise member variables using an initialiser list on constructors, rather than within the body of the constructor.
- Initialize automatic variables at declaration, unless the initial value is dependent on something
- Always put the initializers in the constructor in the same order as they are declared in the class prototype.

Justification

- *Variables in an initialiser list are initialised in the order they are declared in the class rather than the order that they appear in the list.*
- *Un-initialised variables are the cause of a large proportion of bugs.*
- *The initialiser list provides a mechanism for specifying the construction details of members and base classes. Using an initialiser list ensures that members and base classes are initialised to the required values upon their construction, rather than assigning the values later after they have already been default-constructed. It is often more efficient than using assignment within the constructor body.*

Bad Example – In this example m_VarOne will not be correctly initialised as, when it is initialised (first) m_VarTwo won't yet have been set to 5.

```
class CClass1

{

...

CClass1();

...

UINT8 m_VarOne;

UINT8 m_VarTwo;
}

CClass1::CClass1() : m_VarTwo(5), m_VarOne(m_VarTwo) // Error

{

...

}
```

Const

Be Const Correct

- It is recommended that the *const* keyword be used when passing as parameters objects that must not be modified. It takes time to get used to doing, but makes for more robust code.
- Use the earlier placement of const option in complex declarations, where the const is before the type

Examples

```
void f1(const std::string& RName);// pass by reference-to-const

void f2(const std::string* PName);// pass by pointer-to-const

void f3(std::string Name);// pass by value


const int* pPointer // pPointer is a modifiable pointer to a constant int

int* const pPointer // pPointer is a constant pointer to a modifiable int
```

const int* const p // pPointer is a constant pointer to a constant int

No Magic Numbers

- Avoid using magic numbers within source code, i.e. numerical values with no explanation, e.g.

if (foo >= 22)…

- Instead use a well-named constant (including Unit of Measure)
- Simple constants normally place the const early.
- Name should convey why the constant is being used, not what the value is

const UINT32 c_TempThresholdC = 123; ///< Upper Limit in Celsius

if (Temperature >= c_TempThresholdC)…

- Don't create pointless constants such as;

static const UINT32 c_OneThousandMilliseconds = 1000;

better:

static const UINT32 c_SettlingTimeMilliseconds = 1000;

Justification

- *Repeated use of a numeric constant makes it difficult to change the value consistently.*
- *Having a constant describe its value complicates without clarifying*

Control Flow

Condition Expression

- The condition expression in if, for and while must be logical not arithmetic. For example:

```
if (append != 0) // Comment
```

- The fixed size boolean types (e.g. BOOL32) are to be regarded as Boolean and should be used without an arithmetic comparison.

```
BOOL32 MyFlag;

if (MyFlag)

...
```

When Braces are Needed

- All *if*, *else*, *switch*, *while* and *for* statements must have braces
- Braces are required even if the block is empty or if there is only a single statement within them.
- Comment why an empty block is empty
- For example:

```
if (SomeValue == 1)

{

SomeValue = 2;

}
```

Justification

- *It ensures that when someone adds a line of code later there are already braces and they don't forget. It provides a more consistent look. This doesn't affect execution speed. It's easy to do.*

for and while Formatting

- Use loop structures without tricks like endless loops, break or continue whenever possible. The tricks add to complexity and are difficult to maintain.
- It is recognised that endless loops are needed for example for threads.

Recursion

- Avoid recursion, it can generally be transformed to iteration.
- Recursion must never be unconstrained

if then else Formatting

- Use the following layout for If Then Else statements:

```
if (<condition>) // Comment

{

...

}

else if (<condition>) // Comment

{

. . .

}

else // Comment
```

```
{
. . .
}
```

- If you have *else if* statements then it is usually a good idea to always have an else block for finding unhandled cases.

switch Formatting

- Falling through a case statement into the next case statement shall be permitted as long as a comment is included.
- Multiple consecutive case labels that use the same block are not regarded as fall-through and do not need a comment.
- The *default* case should always be present and trigger an error if it should not be reached, yet is reached.
- If you need to create variables put all the case code in a block.

Example

```
switch (...)
{
case 1:
...
// FALL THROUGH

case 2:
{

{
```

```
int Value;

...

}

break;



case 3:

case 4:

...

break;



default:

break;

}
```

Multiple Returns

- A function should have a single point of return.
- Except for precondition checks at the start of a function.

Justification

- *"A single return statement at the end of a function creates a single, known point that is passed through at the termination of function execution. Multiple returns in a single unit should be avoided as much as possible, unless the use of a single return would cause the code to be difficult to understand or maintain." [Source: NASA C++ coding standard]*

- *Embedded returns increase complexity (Essential Complexity / knots) and lead to confusion in maintenance*

Brackets / spaces

White Space

Use white space to divide code into logical segments, to improve readability.

Parentheses *()* with Key Words and Functions Policy

- Do put a space between parentheses and keywords..
- Do put parentheses next to function names.
- Place the parentheses tightly around the contents. Do not insert a space immediately after the opening parenthesis or immediately before the closing one.
- Do use a space on both sides of all binary operators (+,-,*,/,&,|,^,<<,>>,&&,||,etc.)
- Do place unary pointer and reference operators close to the type
- Do place unary +/- next to the operand (no space between).
- Do use parentheses to clarify operator precedence, which may not be clear to all readers
- In particular always use parentheses with bit operators and when mixing arithmetic / bitwise / comparison / logical operators.

- Do not use parentheses in return statements when it's not necessary.

Justification

- *Keywords are not functions. By putting parentheses next to them keywords and function names are made to look alike.*

Example

if (*<condition>*) // space between keyword and bracket

{

}

while (*<condition>*) // space between keyword and bracket

```
{
}
```

```
strcpy(Src, Dest); // Spaces only after each comms
return 1; // No brackets
```

Brace Placement

- Place brace under and inline with keywords:

```
if (<condition>)
{
...
}
else if (<condition>)
{
. . .
}
```

```
else
{
. . .
}

switch (<expression>)
{
e_Option1:
// Code here
break;
e_Option2:
break;
default: // Always include a default
break;
}

while (<condition>)
{
```

```
. . .
} // Reference condition at end of block in long blocks

for ( ; ; )
{
. . .
}

do // Reference condition in long blocks
{
. . .
} while (<condition>);
```

Justification

- *Allows easier matching of opening and closing braces, both visually and when using automated brace matching within some editors.*

Formatting Methods with Multiple Arguments

- Methods with more than three parameters should be formatted as follows:

```
UINT32 AnyMethod(int Arg1, int Arg2,
```

int Arg3, int Arg4);

Tabs

- Indent using 4 spaces for each level.
- **Do not use tabs, use spaces**. All our editors can substitute spaces for tabs, use 4 spaces per tab.

Justification

- *When using tabs, if people use different tab settings the code may be hard to read or may print badly, which is why spaces are preferable to tabs.*

Avoid Schoolboy Errors

Casting

Use the C++ casting operators rather than C style casting

Justification
static_cast *and* reinterpret_cast *provide compile-time type checking. From Stroustrup p130:*
The static_cast *operator converts between related types such as one pointer type to another …an integral type to an enumeration or a floating-point type to an integral type. The* reinterpret_cast *handles conversions between unrelated types such as an integer to a pointer or a pointer to an unrelated pointer type. This distinction allows the compiler to apply some minimal type checking for* static_cast *and makes it easier for a programmer to find the more dangerous conversions represented as* reinterpret_cast*s.*

Don't use const_cast.

Justification
Data is usually const for a reason

Use dynamic_cast with caution

Dynamic cast provides greater type safety, but does require RTTI which is not necessarily supported on all platforms.
Justification
May cause RTTI to be built into the image which increases code size

Error Return Check Policy

- Check every system call for an error return, unless you know you wish to ignore errors. For example, *printf* returns the number of character printed but rarely would you check for it. If you explicitly want to ignore the return code, cast it to void to tell the reader that you've considered and ignored the return code.

Example

```
(void)printf("hello world/n");
```

- Include the system error text for every system error message.

Never Throw Exceptions in Destructors

- Exceptions thrown in a destructor will mean that any subsequent code in the destructor will not be executed. Hence if the subsequent code is intended to free resources this will result in a resource leak.
- Special care must be taken to catch exceptions which may occur during object destruction. Special care must also be taken to fully destruct an object when it throws an exception.

Example

```
// WRONG!

Sql::~Sql(void)

{

delete connection;

delete buffer;// won't be called if connection throws an exception

}

// BETTER

Sql::~Sql(void)

{
```

```
try
{
delete connection;
}
catch (…)
{
… // No 'throw' in here
}
delete buffer;
}
```

Pass large objects by const reference

Don't compare with True

Example

```
// WRONG!
BOOL8 a;
…
```

```
if (a == TRUE)

{

...

}



// BETTER

if (a != FALSE)



// BEST

if (a)
```

Put shared types in a separate types header (namespaced to ModuleTypes) rather than having public typedefs in a class. This reduces class coupling because clients want to use the types don't have to include the class in their header for slots, etc.

Avoid the RAYASSERT(0 == "Error message") construction. Cppcheck rejects it. We should now be using RAYFATAL instead, which achieves the same goal.

Confluence Version History

Version	Date	Author	Comment
31	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
30	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
29	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
28	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
27	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
26	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
25	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
24	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
23	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
22	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
21	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
20	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
19	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
18	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
17	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
16	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
15	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
14	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
13	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
12	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
11	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
10	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
9	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
8	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
7	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
6	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
5	Nov 17, 2017 12:09	Thorne, Andrew	Migration of unmigrated content due to installation of a new plugin
4	Nov 17, 2017 12:09	Thorne, Andrew	Changed colour of main headings to black
3	Nov 15, 2017 16:03	Thorne, Andrew	Added Confluence Version History Macro
2	Nov 10, 2017 16:43	Thorne, Andrew	First Issue from BMS WI1003 v8.0 Dated 01/10/14. Corrected formatting issues on import
1	Nov 10, 2017 16:01	Thorne, Andrew	