REGIONAL ADDITIONAL CODING STANDARDS CPP

QuEST Global Engineering Services Pvt. Ltd.

Copyright Notification	This document is the property of QuEST Global Engineering Services Pvt. Ltd. No part of this document may be reproduced except as authorized by written permission from QuEST Global Engineering Services Pvt. Ltd. The copyright and the foregoing restriction extend to reproduction in all media.
Document Number	TEG-OTH-565
Revision Number	01
Date	01-June-16

Table of Contents

1 Introduction

This document is an annexure to the QuEST Coding Standards and specifies the guidelines to be followed while coding in C++ language. Formatting rules shall be as per the existing QuEST Coding Standards. However, if there are any rules that contradict the one in QuEST Coding Standards, rule detailed in this document will take the priority.

2 General Rules

a. Properly comment whenever a coding standard/rule is not followed. Comment should reflect why this is done, comments should not say rule x is broken; rule is just to remind yourself and reviewer.

b. All rules should be met during coding itself. For example writing the entire code and then formatting/ adding error handling is not a good practice. Approach during coding phase should be "Fix now itself", when it comes to coding standard/rules.

3 Format Related Rules

a. File header, class header and function header shall be as per the QuEST Coding Standards.

b. Function parameter names should match in function declaration and definition.

c. While making a code modification, a brief description of the modification shall be added in the File/Function header and optionally a project specific comment may be added indicating the beginning and end of the modification.

d. Maximum length of line shall be 110 characters, but priority with this respect goes to current project rules.

    i. A suggestion would be to use some kind of a tool/IDE like Visual Assist or Visual Studio which helps us in setting up a vertical marker in the source code editor that will helps us to limit the length of a line.

e. Logically order functions in header and cpp files.

    i. Examples are;

        i. Initialize function should come before Cleanup.

        ii. Get and set functions should be in pair.

    ii. Glaring mismatches should be avoided.

    iii. A complete logical block should be, as much as possible, bunched together.

    iv. Group member functions and variables based on access modifier. An example given below

    v. Maintain the same order for functions and variables declaration and definition.

Example for order of class declaration in a header file

```
class SomeClass
{
// Member functions
public:
      SomeClass();
      virtual ~SomeClass();
      void PublicFunc();

protected:
      void PublicFunctionHelper1();
      void AnotherPublicFuncHelper1();

private:
      void PublicFunctionHelper2();
      void AnotherPublicFuncHelper2();

// Member variables
public:
      // Public variables are not recommented!!!

protected:
      int m_nDataVariable1;

private:
      int m_nDataVariable2;
};
```

f. Opening and closing braces should be in new line and easy to match. To be specific left brace after 'if' should be directly under the first letter of 'if'.

```
if (a != b)
{
    Function();
}
```

g. If more than two nested blocks are in code at the ending brace put a comment about the starting condition.

```
if (Condition1)
{
        while (Condition2)
        {
                if (Condition3)
                {

                }// if (Condition3) ends
        }// while (Condition2) ends
}// if (Condition1) ends
```

h. There should a brace block for control statements even if block has only one line.

```
if (a != b)
{
    Function();
}
```

i. There should not be any space between function parenthesis and parameters [Both in declaration, definition and usage]. If there is more than one parameter, put a space after the comma.

```
int PublicFunc(int nStatus, void* pOutParam);

int SomeClass::PublicFunc(int nStatus, void* pOutParam)
{
        // Function body
}
```

j. Put a space between operators and variables. For Unary operator and cast operator, no need to put the space.

```
a = b + c; // not a=b+c.
++i;
```

k. Name space should also be included for indentation.

```
namespace QuEST
{
        namespace tech
        {
                class TechnicalStreamClass
                {
                };
        }

        class BusinessUnitClass
        {
        };
}
```

l. Put two blank lines between functions in source file

```
// Function Header
void Class1::Function1(void)
{
      statemants;
}


// Function Header
void Class1::Function2(void)
{
      statemants;
}
```

m. Put one blank line between logical blocks in functions

```
QueueID QueueMgr::CreateQ(const char* queName,
                          const int queSize)
{
      QueueID qID = -1;
      int curQId  = m_nextId;

      m_nextId++;

      // Save information related to the new queue
      strcpy(m_queueMap[curQId].name, queName);
      m_queueMap[curQId].id          = m_nextId;
      m_queueMap[curQId].size        = queSize;
      m_queueMap[curQId].front       = -1;
      m_queueMap[curQId].rear        = -1;

      return idCount;
`
```

n. Put one blank line between different access groups (public, protected and private) in header file. See the example in 3(s).

o. Order of variables in header and cpp files should be same for initialization/assignment and copy/= operators.

p. Do not put blank spaces at the end of a line

q. Declare one variable in a line, in class header as well as in function body.

r. If a function has multiple arguments and it cannot be limited to one line it can be continued in next line. In this case, second line should start just after the left "("of the function, and it should start with a variable type.

```
void Employee::PrepareList(CString& csSuccessList,
                           CString& csFailedList,
                           int& nSuccessCount, int& nFailedCount)
```

s. Do not put unnecessary blank lines. See an example for a neatly formatted class.

```
class StatusObject
{
// Member functions
public:
	StatusObject();
	virtual ~StatusObject();
	bool Initialize()
	long GetId();
	void SetId(long lId);
	void Cleanup();

protected:
	CString GetInitializationList();
	bool Unlock(int nLockId);

private:
	bool InitializeTables();

// Member variables
private:
	int m_lId;
	int m_nLockId[LCK_CNT];
};
```

t. Following coding standards are applicable for code generated by framework such as MFC

   i. Provide file header, class header and function header, if code modifications/additions are manually done on that file

   ii. For manually added code, all coding standards are applicable.

   iii. If there is any provision to specify class name, variable name, function name etc before hand in the generated code, then it should follow the naming rules.

4 Comment Related Rules

a. All comments should be in English (do not use any local language word) but these are subject to project requirements.

b. There should be a space between comment symbol and comments text and the comment text should start with Capital letter text.

```
// There should be a space after "//" and comment sentence should
start with CAPITAL letter
```

c. Comments in function header or class header should tell any assumption in function or in class.

d. Comments should not have any spelling mistake or grammar mistake.

e. When a variable is declared put comment above the declaration.

```
// File encryption key
int m_EncKey;
```

f. Recommended comments ratio is 10:3 [Minimum] for source code lines inside functions. Always try to write code in such a way that the source code itself should be self-descriptive.

g. Should not use unnecessary comments. For example; following comment is an unnecessary comment.

```
// Age
int m_Age;
```

h. For inline functions, comments are added in header file, for other functions comments are in cpp file.

i. A short description of interface (public) functions and variables can be provided in header file. Following example gives a short description on public variables

```
public:
      // Length of data, represents number of bytes
      int m_nDataLength;

      // This represent the beauty of Athens
      // inside Olympics  module
      AthensBeatutyClass m_AthensBeatuty;
```

5 Naming Rules

a. Should give a meaningful name for class.

b. To name a global variable use "g_" as prefix (lowercase) and the rest should follow the rule of variable naming. Sample; int g_nErrorNum;

c. To name a member of class use "m_" as prefix (lowercase) and the rest should follow the rule of variable naming. Sample; int m_nEmployeeNum;

d. To name a structure use "_t" as postfix, and the structure type name should be UpperCamelCase.

```
typedef struct _ThreadData
{
      int nParentID;
      long lParam;
}
ThreadData_t;
```

a. To name an enumeration use "_e" as postfix, and the enumeration type name should be UpperCamelCase.

```
typedef enum _FontColor
{
      BLUE = 0,
      GREEN,
      RED,
}
FontColor_e;
```

b. To name a method/function the word order should be Verb+Noun and name should be in UpperCamelCase.

```
bool ReadEmployeeInfo(EmployeeInfo_t* pstEmpInf);

bool WriteEmployeeInfo(EmployeeInfo_t stEmpInf);
```

c. Variable name should be prefixed with lowercase letters to identify the type (System Hungarian notation). Excluding prefix, a variable name should be in UpperCamelCase. General prefixes are defined in the following table:

Type	Prefix	Sample
char	ch	chGrade
char (Array)	sz	szName
TCHAR	tc	tcName
BOOL	b	bEnabled
int	n	nLength
UINT	u	uLength
WORD	w	wPos
long	l	lOffset
unsigned long	ul	ulLength
short	s	sNum
unsigned short	us	usNum
float	f	fValue
byte	by	byTag
struct	st	stStruct
enum	e	eColor
DWORD	dw	dwRange
*	p	pDoc
FAR*	lp	lpDoc
LPSTR	lpsz	lpszName
LPCSTR	lpcsz	lpcszName
LPCTSTR	lpctsz	lpctszName
LPTSTR	lptsz	lptszName
handle	h	hWnd
(*fn)()	lpfn	lpfnAbort
CString	str	csName
double	d	dValue
TCHAR string	tsz	tszName

Table 1: Hungarian notation

e. Use names, which is easy for reading. E.g Prefer PatientArray to ArrayPatient if type is CArray of a Patient class

f. Recommends using suffix '_i' for in-parameters, '_o' for out-parameters and '_io' for in-out-parameters.

6 Control flow Related Rules

a. Do not change a loop variable inside a for loop block.

```
for (int idx = 0; idx < ITEM_COUNT; idx++)
{
        bool status = CalculateAverage(perfIndexList[idx]);
        if (status != STATUS_OK)
        {
                // WRONG! Changing loop variable inside loop
                idx = ITEM_COUNT;
        }
}
```

b. Update loop variables close to where the loop condition is

```
bool isSafeTempLimit = true;

do
{
        // do operations

        isSafeTempLimit = CheckTemeratureLimit();
}
while (inSafeTempLimit);
```

c. All flow control primitives (if, else, while, for, do, switch) shall be followed by a block (putting brace), even if it is empty.

d. All switch statements shall have a default label as the last case label

```
switch (deviceCmd)
{
        case PLAY:
                statements
                break;

        case STOP:
                statements
                break;

        default:
                logError("Invalid command (%d)\n", deviceCmd);
}
```

e. “if-else-if” ladder should have an ending else part.

```
if (condition1)
{
      statements
}
else if (condition2)
{
      statements
}
else
{
      // This block is mandatory for
      // if-else-if ladder
}
```

f. In “if-else-if” ladder do not check for unrelated conditions.

```
// Checking "deviceStatus"
if (deviceStatus == "PLAY")
{
      statements
}
// WRONG! Unrelated condition to "deviceStatus"
else if (tempLimit == 10)
{
      statements
}
else
{
}
```

g. Use bool instead of BOOL, except when an SDK return BOOL instead of bool.

h. Should not use ‘goto’ statements.

i. Do not make explicit comparisons to true or false. It is usually bad style to compare a bool-type expression to true or false.

```
while (condition == false)           // wrong; bad style
while (condition != true)            // also wrong
while ((condition == true) == true)  // where do you stop?
while (booleanCondition)             // OK
```

j. Do not use selection statements (if, switch) instead of simple assignment or initialization. Express your intentions directly.

```
// BAD ONE
bool pos;
if (val > 0)
{
      pos = true;
}
else
{
      pos = false;
}

//better
bool pos = (val > 0) ? true : false;

//better
bool pos = (val > 0); // initialization
```

7 Object Oriented programming Related Rules

a. Avoid member functions that return non-const pointers or references to members.

b. All member variables should have initial value. Initialization list is preferred to assignment. If initialization list is too long move it as assignment with an initialization function which shall be called from the constructor.

c. Always define copy constructor and assignment operator for classes with dynamically allocated memory to its member variables.

d. Make sure that base classes have virtual destructors.

e. Have operator= return a reference to *this. This is to allow operator chaining with minimal overhead.

f. Assign to all data members in operator=. Should do memory allocation for internal pointed data. Also check for self assignment.

g. Data members should not be public.

h. A member function that does not affect the state of an object should be set as const

i. Set function parameters to const, which are not expected to change with in the function.

j. Use const whenever possible.

k. Prefer pass-by-reference to pass-by-value (decision is made by whether it is a simple data type like bool or int, otherwise use const reference)

l. Postpone variable definitions as long as possible. This will minimize scope of the variable and also improves performance depending on control flow.

m. Access functions should be inline

n. Avoid pointers to pointers

o. Use CString to the maximum possible extend if you have included MFC

p. Use STL instead of MFC collection classes, since MFC collection classes were primarily introduced to fill in for lack of support for STL in earlier Microsoft compilers.

q. Header file includes/ forward declaration should be complete and minimal. Definition of classes that are accessed via pointer (*) or reference (&) should not be included as include files, use forward declaration in header files instead.

r. Avoid casts down the inheritance hierarchy.

s. Use C++ -style casts only. Ensure that you applied the proper one (Normally only one we need to use is static cast)
   a. *dynamic_cast Used for conversion of polymorphic types (The value of a failed cast to pointer type is the null pointer, verify Null pointer. A failed cast to reference type throws a bad_cast Exception*
   b. *static_cast Used for conversion of non polymorphic types.*
   c. *const_cast Used to remove the const, volatile, and __unaligned attributes. [Should not be used]*
   d. *reinterpret_cast Used for simple reinterpretation of bits . Use const_cast and reinterpret_cast as a last resort.*

t. Allocate memory only with new and release it with delete, put '[]' for array delete. Recommends using auto_ptr or shared_ptr.

u. Never use memcpy for classes, best example is CString.

v. Name space, use it with scope resolution (Not the using short cut) if the code segment is not very evident. This is mandatory for header files. Another is options is to limit scope of "using" keyword, for e.g. to a function if typing in long namespace name is cumbersome.

w. Use secure functions for string operations.

x. Avoid returning reference/pointer of local objects.

8 Error handling / Exception related Rules

a. Error handling method should be identified for each project at the startup itself.

b. Catch should either propagate or handle it.

c. Final handling point for any "raise" should log, warn user or trace it.

d. Never propagate an exception from Destructor, instead log it and/or display it.

e. Catch exceptions by reference.

f. Constructors can throw exceptions.

g. When you propagate or log an exception make sure the exception location "Classname::Functionname", is put in error string in addition to line number.

9 Miscellaneous Rules

a. Parameter order in function should be - first set should be "in" parameter, then "out" parameter, the "_io" parameter. If default parameter is available use the same at the end of the list.

b. Header includes should use <….h > if the file is a standard file from Windows SDK, MFC. All look up folders are set in VC Environment or in path variables or in Project settings. For project include files "….h" may be used.

c. Project specific includes should not be set as VC Environment or in path variables. Project specific information should go to additional look up folders. Relative path is not expected in source code; instead it should be set in additional look up folders.

d. Avoid the name of the developer inside code comments, except in file header

e. Warning from the files created [Files other than standard include] should be investigated and made zero at compiler warning level 4

f. Function size should not be greater than 100 lines. Ensure that the entire function will be visible in a single screen.

g. Should not hard code anything, use const.

h. Do not access a modified object more than once in an expression.

```
v[i] = ++c; // right
v[i] = ++i; // wrong: is v[i] or v[++i] being assigned to?
i = i + 1; // right
i = ++i + 1; // wrong and useless; i += 2 would be clearer
```

i. Some function like ::create(MFC classes) will have corresponding destroy, use this to avoid resource leakage.

j. ALL win 32 handles should be closed after use.

k. Version numbers are used in File header, function headers, class headers and change description: It has nothing to do with project version / phase. It is in file scope only.

l. All constants definition

    a. If used in the cpp file only, then declare it in cpp itself.

    b. If used in a function only, then declare constant inside the function.

    c. If used by different files, then declare in header file.

m. To avoid the iterative include of the header file, the following should be added in the header file as shown below. Filename.h is the name of this header file, and changes the file name from lowercase to uppercase. Please put "_" between each word. For instance, _AN_BASE_MEDIA_STORAGE_RETURN_H_ for ANBaseMediaStorageReturn.h.

```
#ifndef _FILENAME_H_
#define _FILENAME_H_
       Body of the file
#endif  // ifndef _FILENAME_H_
```

n. **typedef** lengthy template classes. For eg: **typedef CArray<int…> IntArray.** Reduces mistakes and makes it easy to use when specifying this as parameter to function and also as return values.

10 General Rules

a. Do not use magic number or magic string.

b. Domain name (namespace) and Component name should be unique and meaningful. It should reflect the project name. Identify it in AD phase.

c. Boolean function should return true for normal and false for abnormality

d. Error or warning message to user should be done in international (multi national language). If any peculiar error message processing is done, it should be specified in coding guideline of each project.

e. Do not hardcode character sequence, font name, format of date/time, currency, number and address (order of address for e.g. – street number, zip code etc). Exemptions are message to developer and message in the automatically generated code.

f. Check the memory after allocation. Use null check.

g. After deleting a pointer assign it to 0.

10.1 Effective C++

Item	Remarks

1.	Use square brackets for deleting pointer arrays.
	Example –
	string *stringPtr1 = new string;
	string *stringPtr2 = new string[100];
	...
	delete stringPtr1; // delete an object
	delete **[]** stringPtr2; // delete an array of objects

2.	Use delete on pointer members in destructors.
3.	The *new operator* failure condition should be taken care. Check the pointer after allocation.
4.	Adhere to convention when writing operator new and operator delete. When we implement new and delete operator, we should follow the consistent behaviour of the default implementation.
5.	Avoid hiding the "normal" form of new. When we implement operator new other than normal form, we should implement default form also.
6.	Write operator delete if you write operator new.
7.	List members in an initialization list in the order in which they are declared.
8.	Namespace is used positively and the name collision of a global symbol is avoided.
9.	Returning the data (information) which should be hidden in the inside of a class as it stops. (Avoid returning "handles" to internal data.)
10.	Never redefine an inherited non virtual function.
11.	Never redefine an inherited default parameter value.

Table 2: Effective C++

10.2 More effective C++

Item	Remarks
1.	Never treat arrays polymorphically.
2.	Never overload &&, ||, or ,.

Table 3 : More effective C++

10.3 Ellemtel

Item	Remarks
1.	Do not use identifiers (variable name, function name etc) which begin with underscore (`_' or `__').
2.	Always specify the return type of a function explicitly.

Table 4 : Ellemtel

11 C++ coding guideline (recommendation)

11.1 Effective C++

Item	Remarks
1.	Prefer const and inline to #define.
2.	Prefer <iostream> to <stdio.h>.
3.	Prefer new and delete to malloc and free.
4.	Prefer C++-style comments.
5.	Prefer initialization to assignment in constructors.
6.	Avoid data members in the public interface.
7.	Use const whenever possible.
8.	Prefer pass-by-reference to pass-by-value.
9.	Don't try to return a reference when you must return an object.
10.	Choose carefully between function overloading and parameter defaulting.
11.	Avoid overloading on a pointer and a numerical type.
12.	Explicitly disallow use of implicitly generated member functions you don't want.
13.	Avoid member functions that return non-const pointers or references to members less accessible than themselves.

Table 5 : Effective C++

11.2 More effective C++

Item	Remarks
1.	Prefer C++-style casts.
2.	Avoid gratuitous default constructors.

3.	Distinguish between prefix and postfix forms of increment and decrement operators.

Table 6 : More Effective C++

11.3 Ellemtel

Item	Remarks
1.	Avoid using unspecified function arguments (ellipsis notation).
	If a function stores a pointer to an object which is accessed via an argument, let the argument have the type pointer. Use reference arguments in other cases.
2.	Never convert a const to a non-const.

Table 7 : Ellemtel

11.4 Henricson

Item	Remarks
1.	Statements following a case label should be terminated by a statement that exits the switch statement. For fall through cases, add comment /* fall through */
2.	Use break and continue instead of goto.
3.	Literals should only be used in the definition of constants and enumerations.
4.	Avoid unnecessary copying of objects that are costly to copy.
5.	Prefer explicit to implicit type conversions.
6.	Inline function is written as simple as possible.
7.	Do not declare virtual member functions as inline.
8.	Only use a parameter of pointer type if the function stores the address, or passes it to a function that does.
9.	Do not delete ‘this’ pointer.
10.	Document the interface of template arguments.
11.	Pointer should be assigned 0, if an effective value is not assigned to it.
12.	Pointer should be validated before it is referred.
13.	Public method should not return member pointer
14.	Usage of Global variable should be a minimum. Global variable name should reflect the module name
15.	ASSERT/_ASSERTE is used before the condition is processed, should insert as much as possible

Table 8 : Henricson

12 References

I. Effective c++ and More effective C++

II. Programming in C++ Rules and Recommendations – Ellemtel
III. Industrial Strength C++ - Henricson